5. Algorithmes de division.

La division est, on s'en serait douté, la plus difficile et la plus longue à exécuter des quatre opérations arithmétiques de base. Elle est heureusement la moins employée : dans [GZ81], Gosling et Zurawski estiment que dans la plupart des applications courantes scientifiques ou de gestion, les divisions sont au moins dix fois moins fréquentes que les additions/soustractions ou les multiplications.

De même que pour l'addition et la multiplication, les algorithmes les plus simples et les moins coûteux en surface de circuit – mais aussi, hélas, les plus lents – s'obtiennent en transcrivant directement la méthode que nous utilisons lorsque nous effectuons les calculs "à la main".

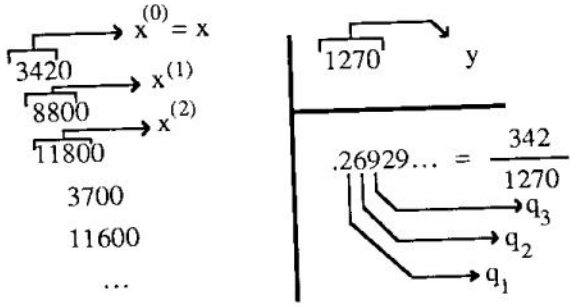
Il y a principalement trois classes d'algorithmes de division, que nous présenterons successivement : tout d'abord ceux utilisant des additions/soustractions et des décalages, ensuite ceux basés sur l'utilisation – tout comme pour la multiplication – de réseaux cellulaires, et finalement ceux qui font appel à des méthodes itératives et à l'utilisation de multiplieurs rapides, comme par exemple la méthode de Newton (itération $x_{n+1} = x_n (2 - a x_n)$).

Lorsque l'on effectue des divisions, il est commode voire nécessaire (il suffit pour s'en convaincre de regarder de près les algorithmes que nous allons bientôt présenter) de pouvoir représenter des nombres négatifs. Par conséquent, nous ne pourrons pas travailler en numération simple de position, et si le contraire n'est pas spécifié, nous supposons que nos opérandes sont écrites en *complément à la base*.

5.1 DIVISION PAR ADDITIONS, SOUSTRACTIONS ET DECALAGES.

Supposons que nous travaillions en base B (en pratique, nous reviendrons vite à B=2), et que nous désirions calculer un quotient x/y, où y sera choisi *positif* (pour simplifier la présentation des algorithmes : le lecteur pourra sans difficulté traiter le cas y<0), et *supérieur strictement à |x|*, afin que x/y soit compris entre –1 et 1.

Examinons tout de suite un exemple : "posons" la division de x = 342 par y = 1270 comme si nous l'exécutions "à la main".



$$x^{(i+1)} = 10\,x^{(i)} - q_{i+1}y$$

Figure 5.1 *Une division effectuée "à la main".*

Notons $x^{(0)} = x$. Nous avons trouvé un entier q_1 pris dans {0, 1, ..., 9} (ici, $q_1 = 1$) tel que $x^{(1)} = 10x^{(0)} - q_1y$ reste positif et inférieur strictement à y. De manière générale, à la $i^{ème}$ étape, nous cherchons un entier $q_{i+1} \in \{0, 1, ..., 9\}$ tel que $x^{(i+1)} = 10x^{(i)} - q_{i+1}y$ reste positif et inférieur strictement à y. Dans ce qui suit, nous allons essayer d'étudier des méthodes semblables, avec d'autres bases de numération et des termes q_i pouvant prendre des valeurs différentes, toutes basées sur des itérations de la forme :

$$x^{(0)} = x$$
$$x^{(i+1)} = Bx^{(i)} - q_{i+1}y$$

où q_{i+1} (pris entre 0 et B–1 si on désire un quotient en numération simple de position, pris entre –B+1 et B–1 si on désire un résultat en "chiffres signés") est choisi tel que $|x^{(i+1)}|$ reste inférieur à y. On obtiendra alors :

$$x^{(1)} = Bx - q_1y$$
$$x^{(2)} = B^2x - (q_1B + q_2)\,y$$

et par récurrence : $x^{(m)} = B^m\,x - (q_1B^{m-1} + q_2B^{m-2} + \ldots + q_m)\,y$

ce qui nous donnera : $q_1B^{-1} + q_2B^{-2} + \ldots + q_mB^{-m} = \frac{x}{y} - B^{-m}\frac{x^{(m)}}{y}$.

Les chiffres $q_1, q_2, q_3, \ldots$ constituent donc une écriture (en numération simple de position ou en "chiffres signés") de x/y. Il y a plusieurs façons de choisir des chiffres q_i répondant aux exigences ci-dessus. Ces différentes façons peuvent s'illustrer par des diagrammes dits *Diagrammes de Robertson*, présentant les différentes valeurs que peut prendre $x^{(j+1)}$ en fonction de $x^{(j)}$. Examinons par exemple les diagrammes ci-dessous (figure 5.2 en base 2, figure 5.3 en base 4). Tout comme dans les chapitres précédents, les entiers négatifs sont notés à l'aide d'une barre (-2 s'écrit $\overline{2}$).

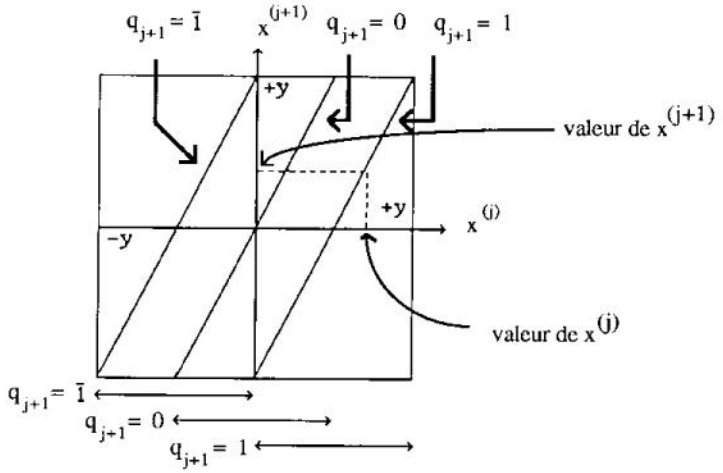


Figure 5.2 *Les différentes possibilités de choix de q_{j+1} et $x^{(j+1)}$ en base 2.*

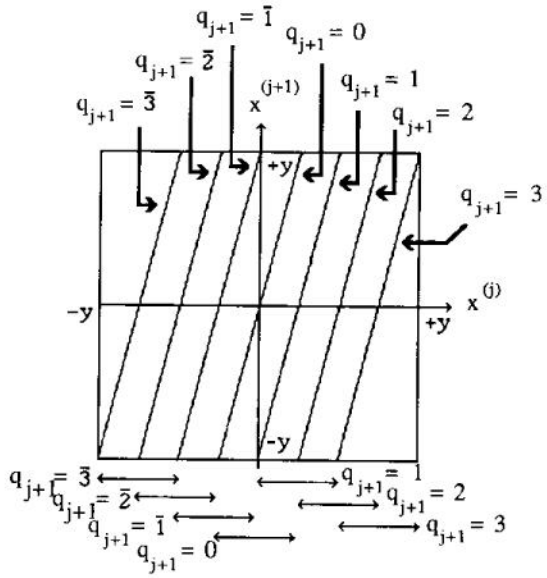


Figure 5.3 *Les différentes possibilités de choix de q_{j+1} et $x^{(j+1)}$ en base 4.*

La figure 5.2 présente les différentes possibilités de choix de q_{j+1} en base 2 :

- si $x^{(j)} < -\ y/2$, il faut choisir $q_{j+1} = \overline{1}$
- si $-\ y/2 \leq x^{(j)} < 0$, on peut choisir $q_{j+1} = \overline{1}$ ou $q_{j+1} = 0$
- si $x^{(j)} = 0$, on peut choisir $q_{j+1} = \overline{1}$ ou $q_{j+1} = 0$ ou $q_{j+1} = 1$
- si $0 < x^{(j)} \leq +\ y/2$, on peut choisir $q_{j+1} = 0$ ou $q_{j+1} = 1$
- si $x^{(j)} > +\ y/2$, il faut choisir $q_{j+1} = 1$

Il y a donc une certaine latitude dans le choix de q_{j+1}. Cette latitude pourra être exploitée dans certains cas, car elle permettra :

– soit de choisir q_{j+1} en n'examinant qu'un petit nombre de bits de $x^{(j)}$, ce qui accélèrera le traitement et réduira le matériel nécessaire (c'est par exemple le cas de la division SRT, que nous étudierons ultérieurement).

– soit de choisir q_{j+1} *avant même* d'avoir fini de calculer $x^{(j)}$. C'est le principe de base des diviseurs "série" et du diviseur cellulaire de Hamacher et Williams.

Cependant, en général, on ne profitera pas de cette latitude de choix, et on imposera une valeur unique de q_{j+1} pour chaque valeur de $x^{(j)}$. Les algorithmes classiques de division peuvent se représenter par un diagramme de Robertson obtenu en "tronquant" les diagrammes ci-dessus (Figures 5.2 et 5.3). Nous allons étudier les plus connus d'entre eux.

5.1.1 La division restaurante.

Les figures 5.4 et 5.5 présentent les diagrammes de Robertson de la division classique (en base 2 pour la figure 5.4, et en base 4 pour la figure 5.5). L'algorithme correspondant est appelé *division restaurante*.

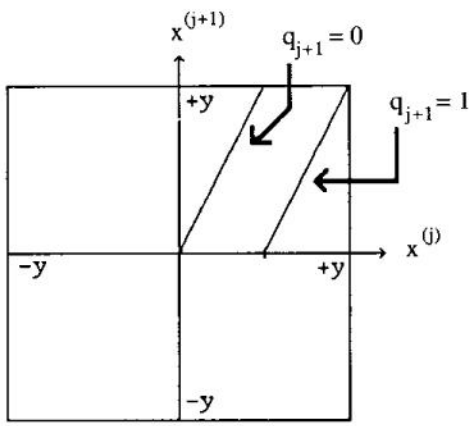


Figure 5.4 *Diagramme de Robertson de la division restaurante en base 2*

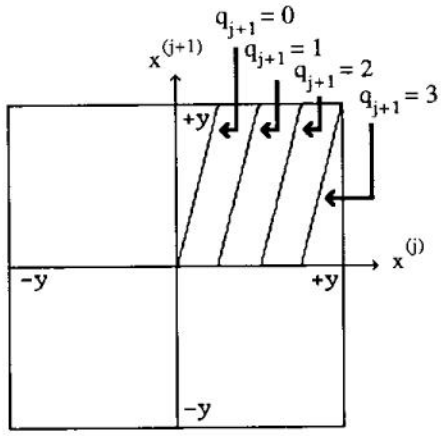


Figure 5.5 *Diagramme de Robertson de la division restaurante en base 4*

La figure 5.4, par exemple, correspond au choix suivant :

- si $x^{(j)} < y/2$, on choisit $q_{j+1} = 0$ et $x^{(j+1)} = 2x^{(j)}$
- si $x^{(j)} \geq y/2$, on choisit $q_{j+1} = 1$ et $x^{(j+1)} = 2x^{(j)} - y$

La figure 5.5, elle, correspond au choix :

- si $y \leq u^{(j)} < 2y$, on choisit $q_{j+1} = 1$ et $x^{(j+1)} = u^{(j)} - y$
- si $y \leq u^{(j)} < 2y$, on choisit $q_{j+1} = 1$ et $x^{(j+1)} = u^{(j)} - y$
- si $x^{(j)} < y/4$, on choisit $q_{j+1} = 0$ et $x^{(j+1)} = 4x^{(j)}$
- si $y/4 \leq x^{(j)} < y/2$, on choisit $q_{j+1} = 1$ et $x^{(j+1)} = 4x^{(j)} - y$
- si $y/2 \leq x^{(j)} < 3y/4$, on choisit $q_{j+1} = 2$ et $x^{(j+1)} = 4x^{(j)} - 2y$
- si $3y/4 \leq x^{(j)}$, on choisit $q_{j+1} = 3$ et $x^{(j+1)} = 4x^{(j)} - 3y$

En pratique, il est plus commode, plutôt que de comparer $x^{(j)}$ aux quantités y/4, y/2 et 3y/4, de calculer $u^{(j)} = 4x^{(j)}$, et de le comparer à y, 2y et 3y.

De manière générale, en base B, la division restaurante correspond aux choix suivants :

- on pose $u^{(j)} = Bx^{(j)}$
- si $u^{(j)} < y$, on choisit $q_{j+1} = 0$ et $x^{(j+1)} = u^{(j)}$
- si $y \leq u^{(j)} < 2y$, on choisit $q_{j+1} = 1$ et $x^{(j+1)} = u^{(j)} - y$
- si $2y \leq u^{(j)} < 3y$, on choisit $q_{j+1} = 2$ et $x^{(j+1)} = u^{(j)} - 2y$

...

- si $(B - 1)y \leq u^{(j)}$, on choisit $q_{j+1} = B\text{-}1$ et $x^{(j+1)} = u^{(j)} - (B - 1)y$

Si on dispose comme primitives de *la multiplication par B* (nous verrons bientôt que même en "complément à la base" une telle opération se fait par un simple décalage) et de *l'addition/soustraction*, alors la division restaurante peut se décrire comme suit.

Division restaurante (base B).
(dividende x, diviseur y, quotient q. On suppose $0 \leq x < y$.*
*$q = (q_1 \ldots q_n)_B$ est rendu en numération simple de position *)*

```
pour i := 1 jusqu'à n faire
    début
     x ← x*B;
     qi ← 0;
     tant que x > 0 faire
        début
            x ← x − y;
            qi ← qi + 1
        fin
     si x < 0 alors
        début
            x ← x + y;
            qi ← qi − 1
        fin
    fin.
```

En base 2, cet algorithme se simplifie considérablement, et devient :

Division restaurante, base 2.

```
pour i := 1 jusqu'à n faire
    début
        x ← x * 2;
        x ← x − y;                    (*)
        si x ≥ 0 alors qi ← 1
        sinon
            début
                qi ← 0;
                x ← x + y             (**)
            fin
    fin.
```

Un rapide examen cet algorithme nous permet de voir d'où vient le terme *restaurante* : une soustraction est effectuée à la ligne (*), et suivant le résultat, soit le résultat obtenu est conservé, soit on *restaure* l'ancienne valeur de la variable x à la ligne (**). Il y a plusieurs façons d'implanter cet algorithme : on peut suivre exac-

tement le canevas décrit ci-dessus et effectuer réellement une addition pour retrouver l'ancienne valeur de x, on peut également mémoriser dans un registre la valeur de cette variable avant d'effectuer la soustraction de la ligne (*), et remplacer la ligne (**) par une lecture dans ce registre.

Exemple.

Supposons que nous travaillions en notation "signe – valeur absolue" en base 2, et considérons la division de $x = (0101)_2 = (5)_{10}$ par $y = (1101)_2 = (13)_{10}$. Nous obtenons les valeurs des termes $x^{(i)}$ et q_i présentées ci-dessous.

i	$x^{(i)}$	$2x^{(i)}$	$2x^{(i)} - y$	q_{i+1}	$x^{(i+1)}$
0	0101	1010	-0011	0	1010
1	1010	10100	0111	1	0111
2	0111	1110	0001	1	0001
3	0001	0010	-1011	0	0010
4	0010	0100	-1001	0	0100

Figure 5.6 *Division restaurante de 5 par 13 en base 2.*

On peut visualiser de manière commode les itérations de la variable $x^{(i)}$ sur un diagramme de Robertson. La figure 5.7 présente la visualisation associée à la même division (quotient de 5 par 13).

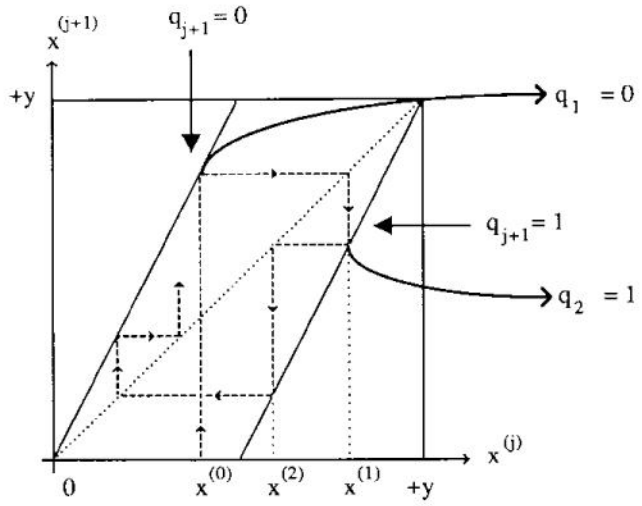


Figure 5.7 *Division restaurante de 5 par 13 en base 2.*

5.1.2 La division non restaurante.

Le terme de *division non restaurante* peut difficilement être évité, car il s'est imposé dans toute la littérature consacrée à la division. Il est toutefois ambigü car si littéralement il désigne tout ce qui n'est pas "division restaurante" (et certains dont Scott [SCO85] utilisent ce terme ainsi), la plupart l'utilisent pour désigner un certain type de division, ou plus exactement certains types, car leurs définitions ne coïncident qu'en base 2. Il m'a donc fallu faire un choix, mais le lecteur devra faire attention lorsqu'il parcourera les articles et livres cités en bibliographie, car ce même terme désigne parfois des choses très différentes. L'algorithme que nous allons étudier fournit un résultat en notation de type "chiffres signés" : si l'on travaille en base B, les chiffres du quotient seront compris entre -B+1 et B-1. Les figures 5.8 et 5.9 présentent les diagrammes de Robertson associés à cet algorithme pour les bases 2 et 4.

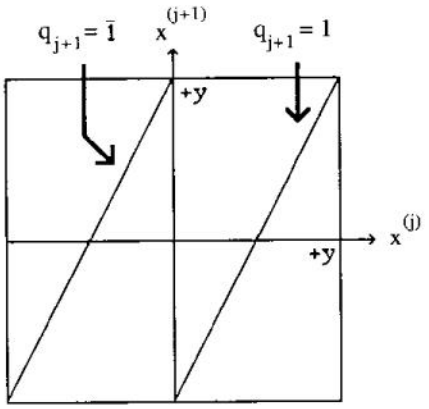


Fig 5.8 *Diagramme de robertson de la division non restaurante en base 2.*

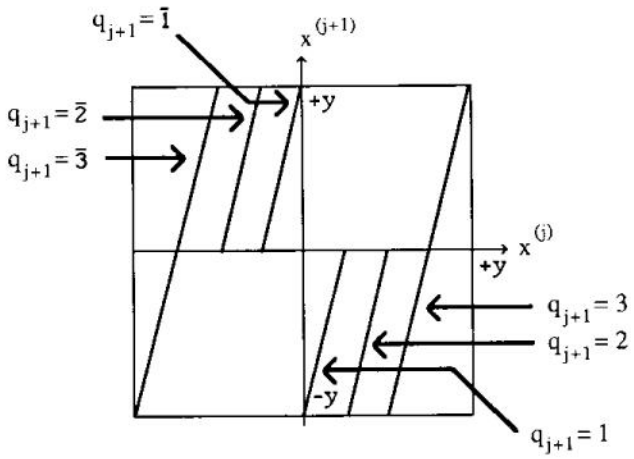


Fig 5.9 *Diagramme de robertson de la division non restaurante en base 4.*

Si on se donne comme primitives l'addition/soustraction et la multiplication par la base de numération B utilisée, la division non restaurante peut se décrire comme suit.

Division non restaurante.
(dividende x, diviseur y, quotient q. $|x| \leq y$.*
*$q = (q_1 \ldots q_n)$ est rendu en "chiffres signés" avec des chiffres pris dans $\{-B+1, -B+2, \ldots, B-2, B-1\}$ *)*

```
pour i := 1 jusqu'à n faire
    début
     x ← x*B
     q_i ← 0
     si x ≥ 0 alors
        tant que (x ≥ 0) et (q_i < B-1) faire
          début
             x ← x - y;
             q_i ← q_i + 1
          fin
     sinon tant que (x ≤ 0) et (q_i > 1-B) faire
          début
             x ← x + y
             q_i ← q_i - 1
          fin
    fin.
```

Tout comme dans le cas de la division restaurante, en base 2, cet algorithme se simplifie considérablement et devient :

Division non restaurante, base 2.

```
pour i := 1 jusqu'à n faire
    début
        x ← x*2
        si (x ≥ 0) alors
            début
                x ← x-y;
                q_i ← 1
            fin
        sinon
            début
                x ← x+y;
                q_i ← 1̄
            fin
    fin.
```

Une remarque s'impose : si nous pouvions travailler en numération usuelle de position – ce qui est hélas impossible puisqu'il nous faut représenter des nombres négatifs –, les affectations "x ← x*B" s'effectueraient par un simple décalage vers les poids forts des chiffres de la représentation de x. Qu'en est-il en complément à la base ? Par construction des algorithmes les termes $x^{(j)}$ sont toujours inférieurs à y, par conséquent, après exécution de "x ← x*B", |x| est toujours inférieur à By. Donc, si on choisit une représentation en complément à la base permettant d'écrire By :

- si x est positif, puisque les écritures en complément à B et en numération simple de position de x et x*B coïncident, effectuer un décalage d'une position vers les poids forts de la représentation de x est rigoureusement équivalent à effectuer "x ← x*B".

- si x est négatif, son écriture en complément à B coïncide avec l'écriture en numération simple de position de B^m+x (où m-1 est le poids du chiffre de poids fort de la représentation adoptée). Si l'on décale cette écriture d'une position vers les poids forts, on obtient un nombre (interprété en numération simple de position) *congru modulo* B^m à $B^{m+1} + B*x$. Puisque B*x est représentable en complément à B dans notre écriture, on obtient en fait $B^m + B*x$, ce que l'on désirait.

Conclusion : En complément à la base, on peut remplacer sans problème les affectations "x ← x*B" par des décalages d'une position vers les poids forts du mot représentant la quantité x.

En base 2, on peut même gagner un bit sur l'écriture des nombres, en supposant seulement que la représentation en complément à 2 utilisée permet d'écrire y (et non plus 2y). En effet, supposons que dans une telle représentation on ait :

$$x = (x_{n-1}x_{n-2} \cdots x_0)_{\overline{2}} = - x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Appelons dec(x) le nombre obtenu en décalant l'écriture de x d'une position vers les poids forts, il vient :

$$\text{dec}\,(x) = (x_{n-2}x_{n-3} \cdots x_0)_{\overline{2}} = - x_{n-2}2^{n-1} + \sum_{i=0}^{n-3} x_i 2^{i+1}$$

on en déduit que 2x-dec (x) est égal à $- x_{n-1}2^n + 2.x_{n-2}2^{n-1} = [x_{n-2} - x_{n-1}]\ 2^n$.

Donc dec (x) est congru à 2x modulo 2^n. Par conséquent, lors de l'éxécution des algorithmes de division restaurante et non restaurante, dec $(x^{(i)}) - q_{i+1}y$ est congru modulo 2^n à $2x^{(i)} - q_{i+1}y$. On en déduit, puisque $2x^{(i)} - q_{i+1}y$ est par construction représentable en complément à 2, que dec $(x^{(i)}) - q_{i+1}y$ est **égal** à $2x^{(i)} - q_{i+1}y$. On

peut donc lors de l'éxécution de ces algorithmes remplacer la multiplication par 2 par un décalage (N.B. dans le cas de la division non restaurante, pour décider des valeurs q_{i+1} des chiffres du quotient, il convient de lire le signe des restes partiels $x^{(i)}$ *avant* de les décaler).

Implantation de la division non restaurante.

Nous allons étudier une petite architecture, semblable à celle de la figure 4.1, permettant d'exécuter l'algorithme de division non restaurante en base 2. Supposons que nos opérandes soient écrites en complément à 2 sur n bits, et que nous désirions calculer les p premiers chiffres du quotient (p peut être supérieur ou inférieur à n). L'architecture présentée ici (figure 5.10) comporte un additionneur/soustracteur de variables écrites sur n bits en complément à 2, qui fournit un résultat écrit lui aussi sur n bits (on rappelle qu'en complément à 2, la retenue sortante est ignorée), deux registres M et ACR de n bits, et un registre ACQ de p bits pour mémoriser le quotient (les chiffres du quotient ne pouvant prendre que deux valeurs $-\overline{1}$ et 1– il suffit d'un bit pour mémoriser chacun d'eux). Nous supposerons que nous disposons d'une instruction DEC qui permet d'effectuer un décalage global d'une position vers les poids forts de l'ensemble (ACQ,ACR) considéré comme un seul registre, et d'une instruction ADD qui effectue les opérations suivantes :

- si le bit de poids faible de ACQ vaut 1, additionner ACR et M, et placer le résultat dans ACR.
- si le bit de poids faible de ACQ vaut 0, soustraire M de ACR, et placer le résultat dans ACR.

On supposera également que l'on dispose d'un compteur C qui peut être décrémenté.

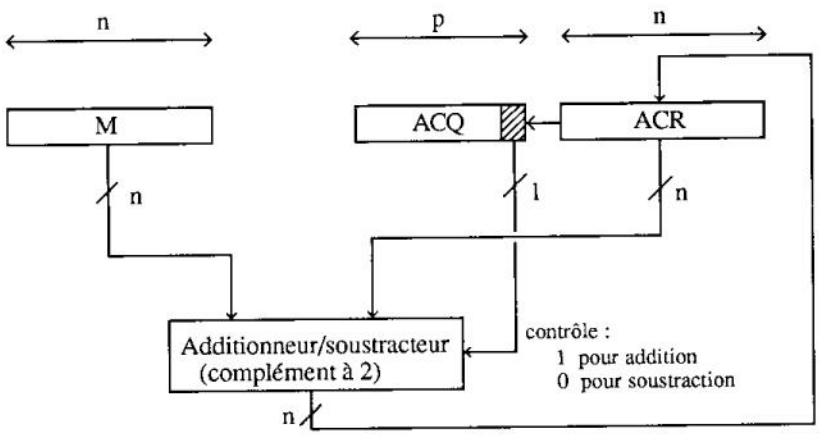


Figure 5.10 *Une architecture de division non restaurante en base 2.*

L'algorithme suivant calcule les p premiers bits du quotient de x par y :

```
début
    placer x dans ACR
    placer y dans M
    C ← p
    tant que C > 0 faire
        début
            DEC
            ADD
            décrémenter C
        fin
fin.
```

Après exécution de cet algorithme, le quotient est mémorisé dans le registre ACQ, et codé comme suit : le chiffre $\bar{1}$ est représenté par 1, et le chiffre 1 est représenté par zéro.

Exemple.
Supposons que nous désirions obtenir sur 6 bits le quotient de $(1101)_{\bar{2}} = -3$ par $(0101)_{\bar{2}} = 5$. La figure 5.11 présente le contenu des registres ACQ et ACR au cours du calcul.

étape	contenu de ACQ après DEC	opération	contenu de ACR après DEC	contenu de ACR après ADD
init.				1101
1	000001	+	1010	1111
2	000011	+	1110	0011
3	000110	–	0110	0001
4	001100	–	0010	1101
5	011001	+	1010	1111
6	110011			

Figure 5.11 *Exemple d'exécution.*

Le résultat final se lit dans le registre ACQ : la chaîne de chiffres (110011) représente les six premiers chiffres du quotient $0.\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\ldots = -\ 0.6 = -\ 3/5$.

Les deux types de division que nous venons d'étudier (divisions restaurante et non restaurante) sont les plus classiques. Il est toutefois fort possible de construire, à l'aide des diagrammes de Robertson, d'autres algorithmes de division. Par exemple, le diagramme de la figure 5.12 nous donne une nouvelle méthode de division en base 2 (elle est quelque peu fantaisiste, mais elle fournit un résultat correct) :

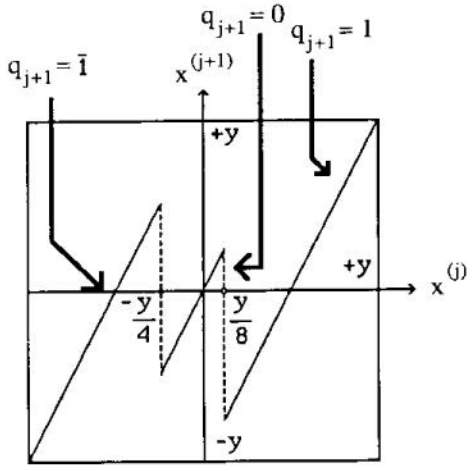


Figure 5.12 *Une division fantaisiste*

Si on utilise un algorithme fournissant des résultats en "chiffres signés" (par exemple l'algorithme de division *non restaurante*), il faudra reconvertir le quotient obtenu en complément à la base. Ceci pourra s'effectuer par le biais d'une simple soustraction (on convertit $(\bar{1}1\bar{1}1\bar{1}1)_2$ en soustrayant $(0110011)_{\bar{2}}$ de $(0001100)_{\bar{2}}$).

5.1.3 La division SRT.

L'algorithme de division SRT (il s'agit en fait d'une classe d'algorithmes, même si ici nous n'en présenterons qu'un seul) doit son nom aux initiales de ses trois inventeurs, D.W. Sweeney, J.E. Robertson ([ROB58], [ROB70]) et K.D. Tocher ([TOC58]), qui semblent l'avoir découvert indépendamment à la même époque, vers la fin des années 50. Nous nous limiterons ici au cas où la base de numération employée est 2, bien qu'une généralisation soit possible. Avant de présenter cet algorithme, cherchons à cerner les avantages et inconvénients respectifs de la division restaurante et de la division non restaurante. Chaque itération de ces algorithmes (ce sera encore le cas de la division SRT) peut se décomposer en deux étapes :

1) Choix du nouveau chiffre q_{i+1} du quotient.

2) Calcul de $x^{(i+1)} = 2x^{(i)} - q_{i+1}y$.

Dans le cas de la division restaurante, l'étape 1 nécessite une comparaison entre $x^{(i)}$ et y, elle est donc assez complexe. Par contre, une fois sur deux en moyenne (on travaille en base 2) q_{i+1} sera nul et l'étape 2 ne nécessitera alors qu'un simple décalage.

Par contre, dans le cas de la division non restaurante, c'est le contraire qui se produit : l'étape 1 s'effectue de manière absolument triviale (q_{i+1} ne dépend que du signe de $x^{(i)}$), par contre puisque q_{i+1} n'est jamais nul, on doit toujours effectuer une addition ou une soustraction à l'étape 2.

Il est honnête de préciser que ces considérations sur l'économie d'une opération lorsque q_{i+1} est nul sont de moins en moins au goût du jour : comme nous l'avons dit lors des chapitres consacrés à l'addition et à la multiplication, lorsque l'on conçoit des opérateurs arithmétiques de base, ce n'est pas le temps **moyen** de calcul qu'il faut chercher à minimiser, mais le **pire** temps.

L'idée fondamentale de la division SRT est de comparer les itérés $x^{(i)}$ non plus à y, mais à une ou plusieurs constantes, **qui s'écrivent sur un petit nombre de bits** (afin d'accélérer les comparaisons). On ne pourra faire ceci de manière efficace que si y est supposé normalisé, c'est-à-dire compris entre deux puissances consécutives de 2 fixées à l'avance. En pratique, on se ramène à un tel cas par des décalages. Cette exigence de normalisation de y n'est en fait pas très contraignante lorsque l'on travaille en virgule flottante, car dans ce cas, x et y sont des mantisses de nombres écrits dans cette notation, et sont donc déjà normalisés. Nous supposerons par la suite :

$$1/2 \leq y < 1$$

$$| x | \leq y$$

L'algorithme de division SRT réside dans le choix suivant de q_{i+1} et $x^{(i+1)}$ en fonction de $x^{(i)}$:

- $x^{(0)} = x$
- $u^{(i)} = 2x^{(i)}$
- si $u^{(i)} < -1/2$ alors $q_{i+1} = \overline{1}$ et $x^{(i+1)} = u^{(i)} + y$
- si $-1/2 \leq u^{(i)} \leq +1/2$ alors $q_{i+1} = 0$ et $x^{(i+1)} = u^{(i)}$
- si $1/2 < u^{(i)}$ alors $q_{i+1} = 1$ et $x^{(i+1)} = u^{(i)} - y$

On peut montrer facilement par récurrence que ce choix nous assure que pour tout i, $| x^{(i)} |$ est inférieur à y. Par conséquent l'algorithme nous fournit bien le quotient désiré, en notation de type "chiffres signés". La figure 5.13 nous présente le diagramme de Robertson de la division SRT pour une valeur de y environ égale à 0.7.

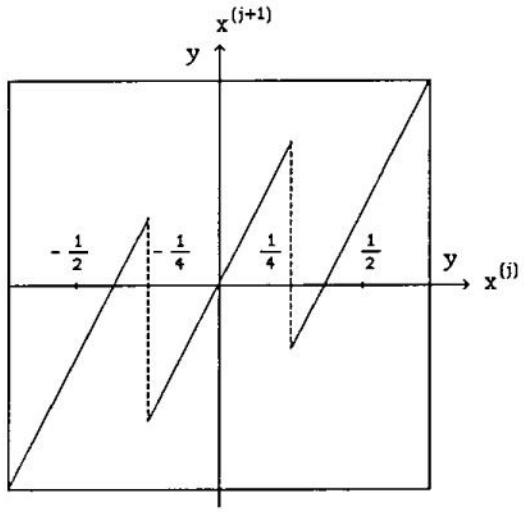


Figure 5.13 *Diagramme de Robertson de la division SRT (base 2, y ≈ 0.7).*

Le grand avantage de la division SRT réside dans la facilité du choix de q_{i+1} à l'étape i, puisqu'une simple comparaison à $\pm 1/2$ suffit, comparaison qui peut s'effectuer en n'examinant que les deux premiers bits de l'écriture de x en complément à 2. De plus, la division SRT fait apparaitre des zéros dans l'écriture du quotient, ce qui évite des opérations arithmétiques lors du calcul de $x^{(i+1)}$ en fonction de $x^{(i)}$. Des études statistiques ont été faites pour essayer d'évaluer la proportion moyenne de zéros créés, mais comme nous l'avons dit précédemment, ce genre de considération tend à devenir de plus en plus obsolète, dans la mesure où il est bien plus cohérent de chercher à minimiser le *pire temps* de calcul : l'utilisation d'un processeur numérique pour lequel le nombre de cycles nécessaires à une division dépendrait de la valeur des opérandes serait très délicate.

5.2 CONSIDERATIONS SUR LA LATITUDE DE CHOIX.

Dans ce paragraphe, nous nous cantonnerons au cas de la base deux. Essayons maintenant de nous attacher au problème suivant : on calcule toujours les chiffres du quotient résultant de la division de x par y à l'aide d'une itération de la forme

$$x^{(0)} = x$$
$$x^{(i+1)} = 2x^{(i)} - q_{i+1}y$$

Mais dorénavant, on suppose que les itérés successifs $x^{(i)}$ sont calculés en notation "carry-save" (voir chapitre 2), afin d'accélérer les calculs. On supposera également que nous ne manipulons que des quantités *normalisées*, vérifiant :

$$0 \leq |x| < y < 1/2$$
$$1/4 \leq y$$

on se ramène à un tel cas par des décalages appropriés. Tous les termes $x^{(i)}$ que nous allons construire seront de valeur absolue comprise entre 0 et y. Cherchons une stratégie intéressante de choix de q_{i+1}. $x^{(i)}$ sera supposé écrit, en notation "carry save" et en complément à 2 sous la forme :

$$x^{(i)} = ((c_0, s_0) , (c_1, s_1) , \ldots , (c_n, s_n)) \quad (1)$$

ce qui signifie que si on note

$$u^{(i)} = \sum_{i=0}^{n} (c_i + s_i)\, 2^{-i}$$

le nombre obtenu en interprétant l'écriture (1) en numération simple de position, et si $u^{(i)}$ s'écrit :

$$(u_{-1}u_0u_1 \ldots u_n)_2 = \sum_{i=-1}^{n} u_i\, 2^{-i}$$

alors :

$$x^{(i)} = - u_0 + \sum_{i=1}^{n} u_i\, 2^{-i}.$$

$x^{(i)}$ s'écrit donc $(u_0u_1 \ldots u_n)_{\bar{2}}$ en complément à deux; il est égal à $(c_0c_1 \ldots c_n)_{\bar{2}}$ + $(s_0s_1 \ldots s_n)_{\bar{2}}$. Donc, pour connaître le signe de $x^{(i)}$ (ou pour effectuer une comparaison entre $x^{(i)}$ et une autre quantité), il nous faut effectuer l'addition :

	c_0	c_1	c_2	...	c_n
+	s_0	s_1	s_2	...	s_n
u_{-1}	u_0	u_1	u_2	...	u_n

et $x^{(i)}$ sera positif si $u_0 = 0$, et négatif sinon.

On comprend qu'il est inutile d'utiliser la notation carry-save si on emploie l'algorithme de division non restaurante, puisque pour connaître le signe de $x^{(i)}$, on est obligé à chaque pas de le transcrire en complément à deux. De même, l'algorithme de division restaurante ou tout autre algorithme faisant appel à une comparaison entre $x^{(i)}$ et une autre quantité peut difficilement être implanté sans réécriture de $x^{(i)}$ en complément à deux : les algorithmes que nous venons d'étudier ne peuvent donc être utilisés de façon efficace.

La latitude de choix des chiffres du quotient va nous tirer de ce mauvais pas. En effet, si au lieu d'effectuer une addition sur n+1 positions dans le but de connaître

exactement l'écriture de $x^{(i)}$ en complément à 2, on effectue une addition sur p+1 positions (p fixe, petit devant n) qui nous fournira une **estimation** de $x^{(i)}$, on peut espérer que cette estimation sera suffisamment précise pour nous permettre de choisir une valeur convenable de $x^{(i+1)}$. Effectuons donc l'addition :

$$\begin{array}{cccccc} & c_0 & c_1 & c_2 & \cdots & c_p \\ + & s_0 & s_1 & s_2 & \cdots & s_p \\ \hline v_{-1} & v_0 & v_1 & v_2 & \cdots & v_p \end{array}$$

La quantité $(v_0 v_1 \ldots v_p)_{\overline{2}}$ est une valeur approchée de $x^{(i)}$. Plus précisément, les p+1 premiers bits $(u_0 u_1 \ldots u_p)$ de l'écriture en complément à deux de $x^{(i)}$ sont égaux soit à $(v_0 v_1 \ldots v_p)$, soit aux bits $(w_0 w_1 \ldots w_p)$ obtenus en effectuant la somme :

$$\begin{array}{cccccc} & v_0 & v_1 & v_2 & \cdots & v_p \\ + & & & & & 1 \\ \hline w_{-1} & w_0 & w_1 & w_2 & \cdots & w_p \end{array}$$

On en tire le résultat suivant :

1) si $v_0 = 1$, et si l'un au moins des v_i $(1 \le i \le p)$ est égal à 0, alors $x^{(i)}$ est forcément négatif. Le choix $q_{i+1} = \bar{1}$ convient donc.

2) si $v_0 = 0$, et si l'un au moins des v_i $(1 \le i \le p)$ est égal à 0, alors $x^{(i)}$ est forcément positif. Le choix $q_{i+1} = 1$ convient donc.

3) si $v_0 = 1$, et si tous les v_i $(1 \le i \le p)$ sont égaux à 1, alors on ne sait rien sur le signe de $x^{(i)}$, mais les p+1 premiers bits de son écriture en complément à 2 sont (11...11) ou (00...00). Sa valeur absolue est donc inférieure ou égale à 2^{-p}.

4) si $v_0 = 0$, et si tous les v_i $(1 \le i \le p)$ sont égaux à 1, alors les p+1 premiers bits de l'écriture en complément à 2 de $x^{(i)}$ sont (0111...11) ou (1000...00), et $x^{(i)}$ est par conséquent de valeur absolue strictement supérieure à 1/2 (et donc à | y |), ce qui est impossible par hypothèse. Ce cas ne se produit donc jamais.

Examinons le cas 3, puisque c'est *a priori* celui pour lequel le choix de q_{i+1} n'est pas immédiat. Rappelons que le diviseur y de la division x/y que l'on désire effectuer est normalisé de sorte que l'on ait $1/4 \le y < 1/2$. D'après le diagramme de Robertson de la figure 5.2, on peut choisir q_{i+1} nul pour $x^{(i)}$ compris entre -y/2 et y/2. Par conséquent, si $x^{(i)}$ est compris entre -1/8 et 1/8, le choix $q_{i+1} = 0$ convient. Dans le cas présent, si on choisit p supérieur ou égal à 3, on peut donc prendre $q_{i+1} = 0$ dans le

cas 3. On obtient alors la stratégie suivante de choix de q_{i+1}, qui ne demande que l'examen des 4 premières positions de l'écriture "carry-save" de $x^{(i)}$, ((c_0, s_0), (c_1, s_1), ... , (c_n, s_n)) .

- Effectuer l'addition

$$\begin{array}{cccccc} & c_0 & c_1 & c_2 & c_3 \\ + & s_0 & s_1 & s_2 & s_3 \\ \hline v_{-1} & v_0 & v_1 & v_2 & v_3 \end{array}$$

(N.B. En pratique, on peut se passer d'effectuer l'addition, voir [HW81])

- Choisir q_{i+1} comme suit :

  – si $v_0 = 0$ et $v_1v_2v_3 = 0$, alors $q_{i+1} = 1$.
  – si $v_0 = 1$ et $v_1v_2v_3 = 0$, alors $q_{i+1} = \bar{1}$.
  – sinon $q_{i+1} = 0$.

5.3 DIVISEURS CELLULAIRES.

Nous allons maintenant essayer d'effectuer des divisions avec des réseaux cellulaires semblables à celui de Braun (voir chapitre 4). Dans tout ce qui suit, nous supposerons que nous travaillons en base 2, et que les opérandes manipulées (diviseur y et dividende x) sont écrites en complément à 2 et en virgule fixe, sous la forme :

$$x = (x_0x_1x_2 \ldots x_{n-1})_{\overline{2}} = -x_0 + \sum_{i=1}^{n-1} x_i 2^{-i}$$

$$y = (y_0y_1y_2 \ldots y_{n-1})_{\overline{2}} = -y_0 + \sum_{i=1}^{n-1} y_i 2^{-i}$$

Nous supposerons que x et y sont **positifs**, et **normalisés** par des décalages appropriés, ce qui implique : $x_0 = y_0 = 0$, $x_1 = y_1 = 1$.

5.3.1 Division cellulaire non restaurante : réseau de Guild.

La division non restaurante est la plus facile à implanter sur un réseau cellulaire. Nous allons ici étudier un réseau dû à Guild ([GUI70]), qui fournit un quotient en "chiffres signés", avec des chiffres valant $\bar{1}$ ou 1. Lors de la $i^{ème}$ étape de l'algorithme de division non restaurante en base 2 de x par y, nous effectuons :

(I) $q_{i+1} = 1$ si $x^{(i)} \geq 0$, $\bar{1}$ sinon.

$x^{(i+1)} = 2x^{(i)} - q_{i+1}y$

Nous supposerons que nous disposons de cellules semblables à celle décrite figure 5.14, et que nous appellerons CELDIV . La cellule "FA" est une cellule élémentaire d'additionneur (voir chapitre 3).

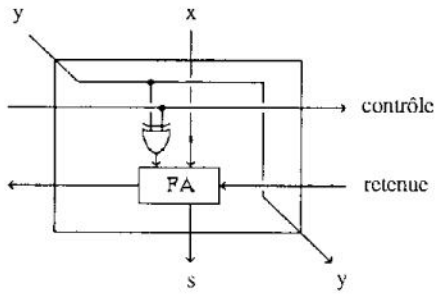


Fig 5.14 Une cellule CELDIV.

Si nous utilisons N cellules CELDIV, disposées en ligne comme suit :

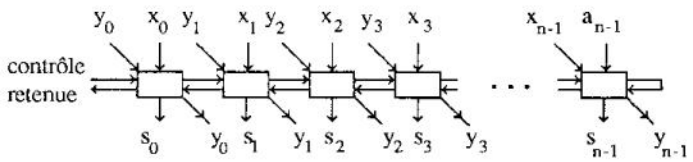


Figure 5.15 *Une ligne de cellules CELDIV.*

puisque nous travaillons en complément à 2, le nombre $(s_0 s_1 \ldots s_{n-1})_{\bar{2}}$ est égal (sauf dépassement de capacité) à la **somme** de $(x_0 x_1 \ldots x_{n-1})_{\bar{2}}$ et $(y_0 y_1 \ldots y_{n-1})_{\bar{2}}$ si la variable "contrôle" vaut 0, et à leur **différence** dans le cas contraire (voir à ce sujet le paragraphe 3.2).

Nous avons déjà vu que les multiplications par 2 qui apparaissent lors de la division non restaurante peuvent être remplacées par des décalages. Notons dec (x) le nombre obtenu en décalant d'une position vers la gauche l'écriture en complément à 2 de x. Le réseau de la figure 5.16, dû à Guild, implante l'itération (I) pour n = 5.

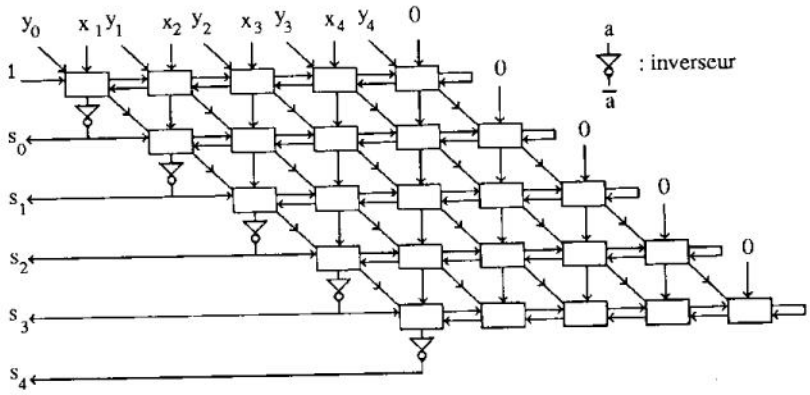


Figure 5.16, *Le réseau de Guild* [GUI70].

En effet, la première ligne de ce réseau reçoit une variable "contrôle" (voir Fig 5.15) égale à 1, elle calcule donc dec($x^{(0)}$)-y, ce qu'il fallait puisque $x^{(0)}$ est, par hypothèse, positif. Le terme s_0 de la figure 5.16 est égal à 1 si et seulement si le bit de poids fort de $x^{(1)}$ = dec($x^{(0)}$)-y est égal à 0, c'est-à-dire si et seulement si cette somme est positive (auquel cas, q_1 est égal à 1, et la deuxième ligne du réseau effectuera une soustraction, ce qui correspond bien à l'itération (I)). De même, la $i^{ème}$ ligne du réseau calcule dec($x^{(i-1)}$) + y^* (avec une retenue entrante égale à "contrôle"), où y^* est égal à y si contrôle = 0, et au nombre obtenu en complémentant les bits de y dans le cas contraire.

Une analyse sommaire de ce réseau permet de comprendre que les termes s_i fournis par le réseau nous permettent de déduire la valeur des chiffres du quotient grâce aux relations :

$$q_0 = 1$$
$$q_{i+1} = 1 \text{ si } s_i = 1, \bar{1} \text{ sinon.}$$

Pour que le réseau de guild fournisse n chiffres du quotient, il faut que le signal de retenue ait traversé les n^2 cellules CELDIV du réseau. Le réseau de Guild effectue donc la division en temps quadratique $O(n^2)$.

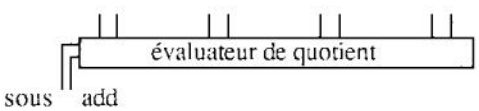
5.3.2 Diviseur de Hamacher et Williams.

De nombreux auteurs ont cru qu'il était impossible d'effectuer des divisions par réseaux cellulaires en un temps meilleur que quadratique (c'est par exemple ce qu'affirme Scott dans [SCO85]). Il est en fait possible dans un premier temps d'effectuer des divisions en temps O(n Log n) en remplaçant les lignes de cellules CELDIV par des additionneurs rapides : une telle solution été proposée par M.

Cappa et V.C. Hamacher dans [CH73], mais elle conduit à des réalisations coûteuses, et qui n'apportent aucune amélioration par rapport aux méthodes de la partie 5.1, qui n'utilisent qu'un seul additionneur rapide.

En 1981, C. Hamacher et J. Williams ont proposé un diviseur cellulaire permettant d'obtenir des quotients sur n chiffres (valant $\bar{1}$, 0 ou 1) en temps O(n). Leur idée de base est somme toute très simple : elle consiste à effectuer les additions/soustractions intermédiaires $x^{(i+1)} = 2x^{(i)} - q_{i+1}y$ en notation *carry save* (Voir chapitre 2), comme dans le multiplieur de Braun présenté au chapitre 4. Le choix de q_{i+1} se fait en utilisant l'algorithme présenté au paragraphe 5.2, qui consiste à n'examiner que les quatre premières positions de l'écriture *carry-save* de $x^{(i)}$.

Le diviseur de Hamacher et Williams est présenté Fig 5.19. L'élément de la figure 5.17 calcule au vu des 4 premières positions de $x^{(i)}$ la valeur de q_{i+1}, qui est codé sur 2 bits, "sous" et "add", de la façon suivante :



si sous = add, alors $q_{i+1} = 0$
si sous = 1 et add = 0, alors $q_{i+1} = 1$
si sous = 0 et add = 1, alors $q_{i+1} = \bar{1}$.

Figure 5.17 *L'élément évaluateur de q_{i+1}.*

La cellule élémentaire du réseau est présentée Figure 5.18. Elle se contente, suivant les valeurs de "add" et "sous" de ne rien faire, de faire une addition ou une soustraction (en "carry save" et en complément à 2 : on transmet les sommes partielles et les retenues, sans les propager, à la ligne suivante, et lorsque l'on soustrait on complémente les bits du nombre soustrait en on introduit une retenue entrante égale à 1).

Il est important de remarquer qu'à chaque instant, une seule ligne de ce réseau est active : on peut donc soit conserver ce réseau complet et effectuer plusieurs divisions en *pipe-line* (mais il ne semble pas qu'il y ait à l'heure actuelle beaucoup de problèmes numériques nécessitant l'emploi d'un diviseur *pipe-line*), soit ne garder qu'une seule ligne de ce diviseur, rebouclée sur elle-même.

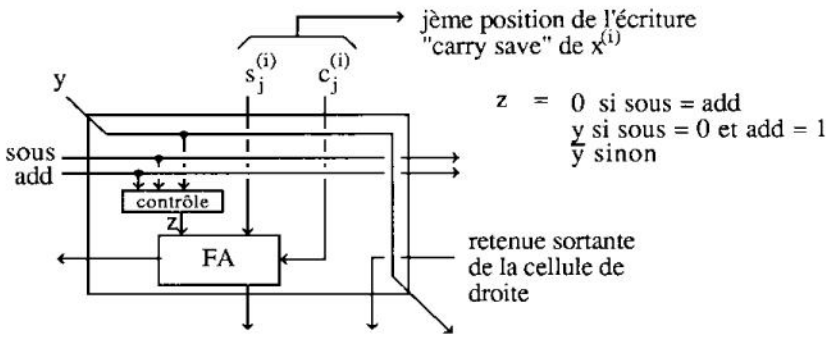


Figure 5.18 *Cellule élémentaire du réseau de Hamacher et Williams.*

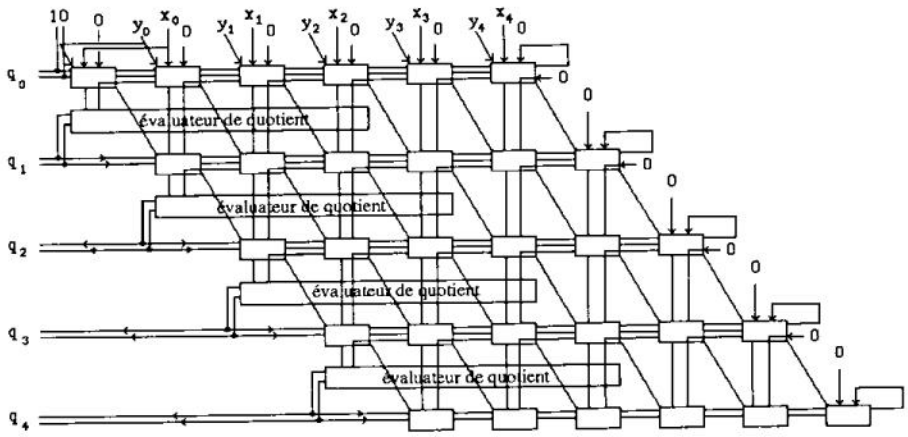


Figure 5.19 *Le réseau de Hamacher et Williams* [HW81].

5.4 DIVISION "EN LIGNE".

Le mode "En ligne" (transmission en série des données *en commençant par les poids forts*) est le seul mode série qui permette d'effectuer des divisions. En effet, même si on utilise un système redondant d'Avizienis pour écrire les nombres, les chiffres de poids faibles d'un quotient dépendent des chiffres de poids fort des opérandes, comme l'illustre l'exemple suivant (base 10, chiffres pris dans {-6, -5, ..., +6}) :

$\frac{4\bar{5}1\bar{6}32}{32}$ admet comme écritures $110\bar{5}1$ et $11\bar{1}51$.

$\frac{3\bar{5}1\bar{6}32}{32}$ admet comme écritures $1\bar{2}\bar{2}26$ et $1\bar{2}\bar{2}3\bar{4}$.

Il est donc impossible de construire un diviseur série pour lequel les chiffres seraient transmis à partir des poids faibles. Bien entendu, on ne peut concevoir des opérateurs "en ligne" qu'en utilisant un système redondant de notation des nombres, au moins pour écrire les résultats, afin d'éliminer les propagations de retenues. Le premier algorithme de division en ligne est dû à Ercegovac et Trivedi [ET77]; par la suite Irwin [IRW78], Irwin et Owens [IO79], [IO87], Ercegovac et Lang [EL85], Tu et Ercegovac [ET87], et Lin et Sips [LS87] ont étudié la question. Nous présenterons ici l'algorithme d'Ercegovac et Trivedi.

Nous noterons les nombres en base 2 et en "chiffres signés", nos opérandes seront :

$$x = (x_1 x_2 \ldots x_n) = \sum_{i=1}^{n} x_i 2^{-i}, \quad x_i \in \{\bar{1}, 0, 1\}$$

$$y = (y_1 y_2 \ldots y_n) = \sum_{i=1}^{n} y_i 2^{-i}, \quad y_i \in \{\bar{1}, 0, 1\}$$

et nous supposerons de surcroît : $\begin{cases} \frac{1}{2} \leq y < 1 \\ |x| \leq y \end{cases}$.

Nos opérandes sont amenées en série. Pour amorcer le processus, nous aurons besoin des 4 premiers chiffres de x et y, c'est-à-dire des quantités :

$$\hat{x}^0 = \sum_{i=1}^{4} x_i 2^{-i} \qquad \hat{y}^0 = \sum_{i=1}^{4} y_i 2^{-i}$$

Par convention, afin de simplifier la présentation de l'algorithme, nous noterons $x_i = y_i = 0$ pour i strictement supérieur à n. L'algorithme d'Ercegovac et Trivedi consiste à construire les chiffres $q_0, q_1, \ldots q_n$ du quotient comme suit.

DIV_LIGNE
début
$\hat{x}^{(0)} := \sum_{i=1}^{4} x_i 2^{-i}$; $\hat{y}^{(0)} := \sum_{i=1}^{4} y_i 2^{-i}$; $q^{(0)} := 0$;
pour j := 0 **jusqu'à** n-1 **faire**
début
si $\hat{x}^{(j)} \geq 1/8$ **alors** $q_{j+1} := 1$
sinon si $\hat{x}^{(j)} \leq -1/8$ **alors** $q_{j+1} := \bar{1}$
sinon $q_{j+1} := 0$;
$q^{(j+1)} := q^{(j)} + q_{j+1} 2^{-j-1}$;
$\hat{y}^{(j+1)} := \hat{y}^{(j)} + y_{j+5} 2^{-j-5}$;
$\hat{x}^{(j+1)} := 2\hat{x}^{(j)} + x_{j+5} 2^{-4} - q_{j+1} \hat{y}^{(j+1)} - q^{(j)} y_{j+5} 2^{-4}$;
fin
fin.

A l'étape j, cet algorithme ne requiert que la connaissance de x_{j+5} et y_{j+5} : il est donc compatible avec une circulation en ligne des chiffres. De plus, comme on travaille en "chiffres signés", toutes les additions peuvent se faire de manière parallèle, sans propagation de retenue, en utilisant l'algorithme d'addition décrit à la fin du chapitre 3.

Montrons maintenant que l'algorithme DIV_LIGNE effectue bien la division de x par y. Une récurrence facile montre que l'on a :

$$(R1) \qquad \hat{x}^{(j)} = 2^j \left[\sum_{i=1}^{j+4} x_i 2^{-i} - \left(\sum_{i=1}^{j} q_i 2^{-i} \right) \left(\sum_{i=1}^{j+4} y_i 2^{-i} \right) \right].$$

Définissons une suite $x^{(j)}$ par : $\begin{cases} x^{(0)} = x \\ x^{(j+1)} = 2x^{(j)} - q_{j+1} y \end{cases}$ *(D)*

où les q_{j+1} sont ceux donnés par DIV_LIGNE. Nous allons essayer de montrer que *(D)* correspond à un algorithme de division en base 2 semblable à ceux étudiés dans la partie 5.1, c'est-à-dire (voir figure 5.2), que l'on a :

- si $x^{(j)} < -\ y/2$, alors $q_{j+1} = \bar{1}$
- si $-\ y/2 \le x^{(j)} < 0$, alors $q_{j+1} = \bar{1}$ ou $q_{j+1} = 0$
- si $x^{(j)} = 0$, alors $q_{j+1} = \bar{1}$ ou $q_{j+1} = 0$ ou $q_{j+1} = 1$
- si $0 < x^{(j)} \le +\ y/2$, alors $q_{j+1} = 0$ ou $q_{j+1} = 1$
- si $x^{(j)} > +\ y/2$, alors $q_{j+1} = 1$

Ce résultat prouverait la correction de l'algorithme DIV_LIGNE.
Il vient sans difficulté :

$$(R2) \qquad x^{(j)} = 2^j \left[\sum_{i=1}^{n} x_i 2^{-i} - \left(\sum_{i=1}^{j} q_i 2^{-i} \right) \left(\sum_{i=1}^{n} y_i 2^{-i} \right) \right].$$

De *(R1)* et *(R2)* on déduit :

$$x^{(j)} - \hat{x}^{(j)} = 2^j \left[\sum_{i=j+5}^{n} x_i 2^{-i} - \left(\sum_{i=1}^{j} q_i 2^{-i} \right) \left(\sum_{i=j+5}^{n} y_i 2^{-i} \right) \right]$$

Soit, puisque les termes x_i, y_i et q_i valent –1, 0 ou 1 :

$$\left| x^{(j)} - \hat{x}^{(j)} \right| \le 2^j \left[\sum_{i=j+5}^{n} 2^{-i} + \left(\sum_{i=1}^{j} 2^{-i} \right) \left(\sum_{i=j+5}^{n} 2^{-i} \right) \right] \le 2^{-3}.$$

Par conséquent :

- lorsque $q_{j+1} = \bar{1}$, $\hat{x}^{(j)} \le -1/8$, donc $x^{(j)} \le 0$

- lorsque $q_{j+1} = 0$, $-1/8 < \hat{x}^{(j)} < +1/8$, donc $-1/4 < x^{(j)} < +1/4$, donc $-y/2 < x^{(j)} < +y/2$, puisque y est compris entre 1/2 et 1
- lorsque $q_{j+1} = 1$, $\hat{x}^{(j)} \geq +1/8$, donc $x^{(j)} \geq 0$.

Donc *(D)* correspond à un algorithme de division en base 2, et par conséquent les termes q_i engendrés par DIV_LIGNE correspondent bien à une écriture à 2^{-n} près de x/y, ce qu'il fallait démontrer.

5.5 DIVISION PAR DES METHODES ITERATIVES.

Plusieurs méthodes itératives ont été proposées pour effectuer la division. Nous présenterons ici les deux plus employées : la méthode de Newton, et une méthode basée sur des produits successifs.

5.5.1 Division par la méthode de Newton.

La méthode de Newton, bien connue des numériciens, consiste, lorsque l'on désire calculer une solution de l'équation $f(x) = 0$ (f est supposée continûment dérivable), à construire une suite (x_n) définie par :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On montre très facilement que si x_0 est suffisamment proche d'une racine simple α de f, alors la suite x_n converge quadratiquement vers α (ce qui signifie grosso-modo que le nombre de chiffres significatifs double à chaque itération).

Nous allons calculer l'inverse d'un réel positif a. Dans ce but, nous chercherons l'unique racine de $f(x) = 1/x - a$. Puisque $x - f(x) / f'(x)$ est égal à $x.(2-ax)$, l'itération de Newton associée à notre problème est : $x_{n+1} = \Phi(x_n) = x_n (2 - ax_n)$. Dans le but de savoir quel point de départ x_0 adopter pour que cette itération converge, il convient d'étudier la fonction Φ. Examinons sa représentation graphique (figure 5.20)

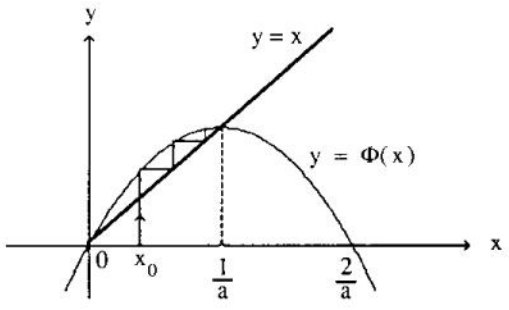


Figure 5.20 *Itération de Newton associée à la division.*

On voit clairement sur la figure précédente que l'itération de Newton converge si et seulement si x_0 appartient à] 0 , 2/a [. Nous nous trouvons face à un problème qui peut paraître insoluble : pour trouver le point x_0 qui permettra de calculer 1/a, il nous faut apparemment connaître 2/a !

Heureusement, lorsque l'on travaille en virgule flottante, les mantisses des nombres sont normalisées : si a s'écrit $m.B^e$ ($m \in [1/B,1[$), alors 1/a sera égal à $1/m.B^{-e}$. Il suffit donc d'être capable de calculer 1/m par la méthode de Newton. Or, puisque m appartient à [1/B , 1 [; 1/m appartient à] 1 , B] et 2/m appartient à] 2 , 2B], par conséquent tout point de départ x_0 inférieur à 2 conviendra (on peut par exemple prendre x_0 égal à 1). En pratique on peut faire beaucoup mieux en mémorisant dans une table les inverses de tous les nombres de [1/B, 1 [qui s'écrivent sur un petit nombre p de chiffres (4 chiffres en base 2, par exemple), et en choisissant comme point de départ pour calculer 1/a l'inverse, lu dans la table, du nombre obtenu en ne gardant que les p premiers chiffres de a : la méthode de Newton étant quadratique, au bout de i itérations, on connaitra 1/a avec environ $p.2^i$ chiffres.

Exemple.

Supposons que nous travaillons en base 10, et cherchons à calculer $Q = 0.42\ 10^{56} / 0.37\ 10^{-21}$ à l'aide de cette méthode. Nous allons tout d'abord chercher l'inverse de 0.37. Si nous partons du point de départ $x_0 = 1$, les itérés successifs de la méthode de Newton ($x_{i+1} = x_i\ (2–0.37x_i)$), sont :

$$x_1 = 1.63$$
$$x_2 = 2.276947$$
$$x_3 = 2.635633573$$
$$x_4 = 2.701038343$$
$$x_5 = 2.702701678$$
$$x_6 = 2.702702702$$

Le quotient recherché est $(2.702702702.10^{21})x(0.42.10^{56}) = 1.135135135.10^{77}$.

5.5.2 Une autre méthode quadratique.

Cette méthode, dont l'auteur ne semble pas connu, a été présentée dans [AEGP67], [FLY70], [HWA79] et [SCO85]. Elle est quadratique tout comme la méthode de Newton mais elle est probablement plus aisée à mettre en oeuvre, et a été implantée dans l'IBM System/360 model 91 ([AEGP67]).

Supposons que nous ayons à calculer un quotient Q = N/D, où N et D sont supposés positifs et normalisés (on les prendra compris entre 1/B et 1, B étant la base dans laquelle on travaille). Nous allons essayer de construire des termes R_0, R_1, ..., R_i choisis de sorte que $D{\cdot}R_0{\cdot}R_1{\cdot}R_2 \ldots R_i$ converge le plus vite possible vers 1. Il viendra alors de manière évidente :

$$N \cdot R_0 \cdot R_1 \cdot R_2 \ldots R_i \longrightarrow Q$$
$$i \longrightarrow \infty$$

Posons $\varepsilon = 1-D$. Puisque D est normalisé, ε est compris entre 0 et $1 - 1/B$. Si nous choisissons $R_0 = 1 + \varepsilon$, DR_0 sera égal à $(1-\varepsilon)(1+\varepsilon) = 1-\varepsilon^2$; si maintenant nous choisissons $R_1 = (1 + \varepsilon^2)$, DR_0R_1 sera égal à $1-\varepsilon^4$.

Le lecteur se persuadera aisément (la récurrence est immédiate) que si l'on pose $R_i = (1 + \varepsilon^{2^i})$, alors $D \cdot R_0 \cdot R_1 \cdot R_2 \ldots R_i$ est égal à $(1 - \varepsilon^{2^{i+1}})$: nous avons donc bien obtenu la convergence désirée, elle est de plus quadratique.

Cette méthode est particulièrement efficace en base 2, car ε est inférieur ou égal à 1/2, ce qui assure une convergence très rapide. Il est bien évident qu'en pratique on ne perd pas son temps à calculer les termes $D \cdot R_0 \cdot R_1 \cdot R_2 \ldots R_i$. Voici l'algorithme :

Division Quadratique.
(N et D sont supposés normalisés *)*
(résultat : le quotient Q *)*

début
 $\varepsilon \leftarrow 1-D$;
 $Q \leftarrow N$;
 pour i := 0 **jusqu'à** K **faire**
  début
   $Q \leftarrow Q (1 + \varepsilon)$;
   $\varepsilon \leftarrow \varepsilon^2$
  fin
fin.

L'éxécution de cet algorithme nous assurera une erreur relative sur le quotient Q inférieure ou égale à :

$$\left(1 - \frac{1}{B}\right)^{2^{(K+1)}}$$

en particulier, si B est égal à deux, cette majoration d'erreur devient égale à $2^{-(2^{K+1})}$

Exemple.

Cherchons à effectuer par cette méthode la division (en base 10) de N = 0.2 par D = 0.7. Il vient :

$$\varepsilon = 0.3$$
$$N.R_0 = 0.26$$
$$N.R_0.R_1 = 0.2834$$
$$N.R_0.R_1.R_2 = 0.28569554$$
$$N.R_0.R_1.R_2.R_3 = 0.2857142845$$
$$N.R_0.R_1.R_2.R_3.R_4 = 0.2857142857,$$

ce qui nous donne bien 2/7 avec 10 chiffres significatifs.

Mais en fait... cette méthode n'est pas très originale ! En effet, reprenons l'itération associée à la méthode de Newton : $x_{n+1} = x_n\,(2 - ax_n)$, on peut l'écrire : $x_{n+1} = x_n + x_n\,(1 - ax_n)$.

Si nous posons $\varepsilon_n = (1 - ax_n)$, il vient presque immédiatement :

$$\begin{cases} x_{n+1} = x_n\,(1 + \varepsilon_n) \\ \varepsilon_{n+1} = \varepsilon_n^2 \end{cases}$$

Et nous retombons sur la méthode précédente, qui n'est donc qu'une reformulation de la méthode de Newton avec un point de départ x_0 égal à un !

BIBLIOGRAPHIE

[AEGP67] S.F. Anderson, J.G. Earle, R.E. Goldschmidt et D.M. Powers, *The IBM system/360 model 91 : Floating-point execution unit*, IBM Journ. of research and development, Janvier 1967.

[AGR79] D.P. Agrawal, *High speed arithmetic arrays*, IEEE Transactions on Computers, Vol. C-28, pages 215-224, Mars 1979.

[ATK68] D.E. Atkins, *Higher-Radix division using estimates of the divisor and partial remainders*, IEEE Transactions on Computers, Vol. C-17 N° 10, Octobre 1968.

[ATK70] D.E. Atkins, *Design of the arithmetic units of ILLIAC III : use of redundancy and higher radix methods* , IEEE Transactions on Computers, Vol. C-19 N° 8, Aout 1970.

[BPPT87] B.K. Bose, D.A. Patterson, L. Pei et G.S. Taylor, *Fast multiply and divide for a VLSI Floating-Point unit*, Actes du 8^{th} *Symposium on computer arithmetic*, (Come, Italie, Mai 1987), Publication IEEE N° 87CH2419-0.

[BUS83] L.B. Bushard, *A minimum table size result for higher radix nonrestoring division*, IEEE Transactions on Computers, Vol. C-32 N° 6, Juin 1983.

[CAV84] J.J.F. Cavanagh, *Digital computer arithmetic, design and implementation* , Mc Graw-Hill Computer Science Series, 1984.

[CH73] M. Cappa et V.C. Hamacher, *An augmented iterative array for high-speed binary division*, IEEE Transactions on Computers, Vol. C-22, Février 1973.

[EL85] M.D. Ercegovac et T. Lang, *A division algorithm with prediction of quotient digits*, Actes du 7^{th} *Symposium on computer arithmetic*, Urbana, Illinois, Juin 1985.

[EL87] M.D. Ercegovac et T. Lang, *On-the-fly conversion of redundant into conventional representations*, IEEE Trans. on Computers, Vol. C-36 N° 7, Pages 895-897, Juillet 1987.

[ERC84] M.D. Ercegovac, *On-line arithmetic : an overview*, SPIE Vol. 495, Real Time Signal Processing VII, pages 86-93, 1984.

[ET77] M.D. Ercegovac et K.S. Trivedi, *On line algorithms for division and multiplication*, IEEE Transactions on Computers, Vol. C-26, pages 681-687, Juillet 1977.

[ET87] M.D. Ercegovac et P.K.G. Tu, *A radix-4 on-line division algorithm*, Actes du 8^{th} *Symposium on computer arithmetic*, (Come, Italie, Mai 1987), Publication IEEE N° 87CH2419-0.

[FAN87] J. Fandrianto, *Algorithm for high speed shared radix 4 division and radix 4 square root*, Actes du 8^{th} *Symposium on computer arithmetic*, (Come, Italie, Mai 1987), Publication IEEE N° 87CH2419-0.

[FER67] D. Ferrari, *A division method using a parallel multiplier*, IEEE Transactions on Computers, Vol. EC-16, Avril 1967.

[FLY70] M.J. Flynn, *On division by functional iteration*, IEEE Transactions on Computers, Vol. C-19, Aout 1970.

[GE80] A.L. Grnarov et M.D. Ercegovac, *On the performance of on-line arithmetic*, Proc. 1980 Intern. Conference on parallel processing, IEEE Publ. N° 80CH1569-3, pages 55-62, Aout 1980.

[GH71] A.B. Gardiner et J. Hont, *Comparison of restoring and nonrestoring cellular array dividers*, Electronic letters, Vol. 7, Juillet 1971.

[GH73] J. Gavilan et V.C. Hamacher, *High-Speed multiplier/divider iterative arrays*, Proc. of 1973 Sagamore computer conference on parallel processing, 1973, pages 91-100.

[GPS87] M. Gavrielov, V. Peng et S. Samudrala, *On the implementation of shifters, multipliers and dividers in VLSI floating points units*, Actes du 8^{th} *Symposium on computer arithmetic*, (Come, Italie, Mai 1987), Publication IEEE N° 87CH2419-0.

[GUI70] H.H. Guild, *Some cellular logic arrays for nonrestoring binary division*, Radio Electron. Eng., Vol. 39, 1970, pages 345-348.

[GZ81] J.B. Gosling et J.H.P. Zurawski, *Design of High-Speed digital divider units*, IEEE Transactions on Computers, Vol. C-30, N° 9, Septembre 1981.

[HWA79] K. Hwang, *Computer arithmetic principles, architecture and design* , New-York, J. Wiley&Sons Inc., 1979.

[HW81] V.C. Hamacher et J. Williams, *A linear-time divider array*, Canadian Electr. Engineering Journal, Vol6, N° 4, 1981.

[IO79] M.J. Irwin et R.M. Owens, *On-line algorithms for the design of pipeline architectures*, 6^{th} symposium on Computer Architecture, Philadelphia, PA, Avril 1979.

[IO87] M.J. Irwin et R.M. Owens, *Digit-pipelined arithmetic as illustrated by the paste-up system : a tutorial*, IEEE Computer, pages 61-73, Avril 1987.

[IRW78] M.J. Irwin, *A pipelined processing unit for on-line division*, Proc. 5^{th} symposium on Computer architecture, IEEE Publ. N° 78CH1284-9C, pages 24-30, Avril 1978.

[KNU69] D.E. Knuth, *The art of computer programming* , Vol.2, *Semmi-numerical algorithms*, Addison Wesley, Reading, Mass., 1969.

[KRI70] E.V. Krishnamurthy, *On range transformation techniques for division*, IEEE Transactions on Computers, Vol. C-19, N° 3, Mars 1970.

[LS87] H. Lin et H.J. Sips, *A novel floating-point online division algorithm*, Actes du 8^{th} *Symposium on computer arithmetic*, (Come, Italie, Mai 1987), Publication IEEE N° 87CH2419-0.

[MAJ70] J.C. Majithia, *Nonrestoring binary division using a cellular array*, Electronic letters, Vol. 6, Mai 1970.

[MC80] C. Mead et L. Conway, *Introduction to VLSI systems* , 1980, Ed. Addison-Wesley.

[MET62] G. Metze, *A class of binary divisions yielding minimally represented quotients*, IRE Transactions on Electronic Computers, Vol. EC-11 N° 6, Décembre 1962.

[ROB58] J.E. Robertson, *A new class of digital division methods*, IRE Transactions on electronic computers, Vol. C-7, Septembre 1958.

[ROB70] J.E. Robertson, *The correspondence between methods of digital division and multiplier recoding procedures*, IEEE Transactions on Computers, Vol. C-19 N° 8, Aout 1970.

[SCO85] N.R. Scott, *Computer systems and arithmetic* , Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985.

[SIP84] H.J. Sips, *Bit-sequential arithmetic for parallel processors*, IEEE Transactions on Computers, Vol. C-33 N° 1, Janvier 1984.

[SPA81] O. Spaniol, *Computer arithmetic : logic and design*, New York, Wiley&Sons, 1981.

[STE72] R. Stefanelli, *A suggestion for High-Speed parallel binary divider*, IEEE Transactions on Computers, Vol. C-21, Janvier 1972.

[TOC58] K.D. Tocher, *Techniques of multiplication and division for automatic binary computers*, Quart. Journ. Mech. Appl. Math., Vol.11, pt 3, 1958, pages 364-384.

[TUN68] C. Tung, *A division algorithm for signed-digit arithmetic*, IEEE Transactions on Computers, Vol. C-17, 1968.

[TUN70] C. Tung, *Signed-digit division using combinational arithmetic nets*, IEEE Transactions on Computers, Vol. C-19 N°8, Aout 1970.

[WF82] S. Waser et M.J. Flynn, *Introduction to arithmetic for digital systems designers*, Holt, Rinehart & Winston, CBS College publishing, New York, 1982.

6. Calcul des fonctions élémentaires.

INTRODUCTION

Le calcul des fonctions mathématiques usuelles (fonctions trigonométriques, exponentielle, logarithme, etc.) constitue la base de tout le calcul scientifique. Il y a encore peu de temps, l'implantation de ces fonctions ne se concevait que de manière *logicielle*, et la grande majorité des ordinateurs calcule encore ces fonctions par programme; cependant, de nos jours, les possibilités sans cesse croissantes de l'intégration sur silicium nous autorisent à envisager des réalisations *matérielles*. La différence considérable entre ces deux modes d'implantation induira inévitablement une différence également très grande entre les algorithmes utilisés : les méthodes logicielles traditionnelles pour le calcul de ces fonctions, basées généralement sur des approximations par des *polynômes* ou des *fractions rationnelles* ne peuvent que difficilement s'appliquer en matériel, car elles font appel à des multiplications et des divisions en virgule flottante sur un grand nombre de bits, ce qui peut être coûteux en surface de silicium. Toutefois, dans [DM88], Duprat et Muller présentent un opérateur spécialisé, un *polynômieur*, susceptible d'être placé sur un circuit intégré, et capable d'évaluer rapidement des polynômes : avec un tel opérateur, les méthodes de calcul des fonctions élémentaires basées sur des approximations polynômiales de ces fonctions reprennent tout leur intérêt en matériel. Nous étudierons tout d'abord succintement ces méthodes traditionnelles.

Par la suite, nous nous appesantirons sur des méthodes différentes, basées sur des primitives autres que la multiplication et de la division. Nous présenterons ici des algorithmes n'employant que l'*addition* et la *multiplication par une puissance entière de la base de numération employée*, opération qui se réduit à un *décalage* si l'on travaille en virgule fixe, et à une *addition à l'exposant* si l'on travaille en virgule flottante (par abus de notation, nous conviendrons d'appeler dans tous les cas cette opération un *décalage)*.

6.1 METHODES CLASSIQUES.

Les opérations de base dont nous disposons en machine sont les additions, les soustractions, les multiplications, les divisions (pas toujours !) et les comparaisons. Il n'est pas difficile de constater que l'ensemble des fonctions d'une variable que l'on peut calculer à partir de ces primitives est l'ensemble des *polynômes* par morceaux (si on n'admet pas la division), et l'ensemble des *fractions rationnelles* par morceaux (si on l'admet).

Si on désire calculer en machine une fonction f quelconque, nous serons donc obligés de l'approcher par des fonctions polynômiales ou rationnelles par morceaux. Un théorème, dû à Weierstrass (voir plus loin), nous apprend que toute fonction *continue* peut être approchée d'aussi près qu'on le désire, *sur un intervalle borné*, par un polynôme (et donc également par une fraction rationnelle). Si nous désirons calculer une fonction sur tout son domaine de définition (qui n'est en général pas borné), il faudra donc effectuer trois étapes :

– La *réduction d'argument* qui consiste à se ramener à l'intervalle dans lequel l'approximation rationnelle ou polynômiale est valide. Elle consiste à déduire de l'argument d'origine x un *argument réduit* y inclus dans l'intervalle en question, et tel que f(x) puisse être déduit de f(y). Par exemple, si l'on désire calculer le sinus de 1000, et si l'intervalle dans lequel on possède une approximation de la fonction sinus est $I = [-\pi/2, \pi/2]$, alors l'argument réduit y associé à 1000 sera l'élément de I, obtenu à l'aide d'une simple division, qui peut s'écrire sous la forme $y = 1000 + N\pi$, N entier, soit $y = 0.973\ldots$

– Le calcul de la valeur que prend l'approximation de la fonction pour l'argument réduit. Dans le cas de l'exemple précédent, cette opération revient à approximer $\sin(y) \approx 0.826\ldots$

– La *restitution d'argument* qui consiste à déduire f(x) de f(y). Dans l'exemple traité, puisque le nombre N tel que $y = 1000 + N\pi$ est pair (N = 318), le sinus de x est égal au sinus de y.

Nous donnerons tout d'abord quelques résultats classiques concernant le calcul d'approximations polynômiales ou rationnelles d'une fonction continue sur un intervalle borné, nous consacrerons ensuite quelques lignes à la méthode de Newton.

6.1.1 Approximations polynômiales.

Notons P_N l'ensemble des polynômes de degré inférieur ou égal à N. Nous désirons approcher, sur l'intervalle [a , b], une fonction continue f par un élément p^* de P_N. Il est important de noter qu'une approximation de f par son développement de Taylor (s'il existe !) en un point de I est en général peu satisfaisante car le développement de Taylor fournit une approximation *locale*, mais non point *globale*.

Nous chercherons ici, pour une norme fonctionnelle ||.|| donnée, à minimiser ||f-p||, $p \in P_N$. Deux critères d'approximation seront traités ici : l'approximation au sens des *moindres carrés* et l'approximation au sens de la norme de la *convergence uniforme*.

Meilleure approximation polynômiale au sens des moindres carrés.

Nous cherchons un élément p^* de P_N vérifiant :

$$\int_a^b w(x)\,[f(x) - p^*(x)]^2\,dx = \min_{p \in P_N} \int_a^b w(x)\,[f(x) - p(x)]^2\,dx$$

où w est une *fonction poids* positive et intégrable sur [a,b].

Puisque la norme $\| f \| = \left(\int_a^b w(x)\, f^2(x)\, dx \right)^{1/2}$ est associée au produit scalaire :

$$< f , g > = \int_a^b w(x)\, f(x)\, g(x)\, dx$$

il suffit d'effectuer un simple calcul de projection. La marche à suivre est donc la suivante :

– en appliquant le procédé d'orthogonalisation de Schmidt à la famille $(X^p)_{p \le N}$, on construit une *base orthogonale* $(T_p)_{p \le N}$ de polynômes pour le produit scalaire $< f , g >$ telle que T_p est de degré p.

– on calcule les coefficients $a_i = \dfrac{< f , T_i >}{< T_i , T_i >}$.

– on obtient $p^* = \sum_{i=0}^{N} a_i T^i$.

Le point crucial de cette méthode est le calcul des coefficients a_i. Ce calcul peut se faire analytiquement (avec éventuellement l'utilisation d'un système de *calcul formel*) si les fonctions f et w sont suffisamment simples, mais si ce n'est pas le cas, il faut faire appel à un calcul numérique précis, en se servant d'un logiciel *multiprécision*.

Certaines familles de polynômes orthogonaux, associées à des fonctions poids w(x) simples, sont connues depuis longtemps, nous allons maintenant présenter les plus classiques.

Polynômes de Legendre. $(w = 1 , a = -1 , b = 1)$

Ils sont définis par : $\begin{cases} p_o(x) = 1 \\ p_1(x) = x \\ p_n(x) = \dfrac{2n-1}{n}.x.p_{n-1}(x) - \dfrac{n-1}{n}p_{n-2}(x) = \dfrac{1}{n!.2^n}\dfrac{d^n}{dx^n}[(x^2 - 1)^n] \end{cases}$

Ces polynômes vérifient : $\langle p_i , p_j \rangle = \begin{cases} 0 \text{ si } i \neq j \\ \dfrac{2}{2i + 1} \text{ sinon} \end{cases}$

Polynômes de Chebyshev. $\left(w(x) = \dfrac{1}{\sqrt{1 - x^2}} , a = -1 , b = 1\right)$

Ils sont définis par : $T_n(x) = \cos [n \text{ Arc cos } (x)]$, ou par les relations de récurrence :

$$\begin{cases} T_o(x) = 1 \\ T_1(x) = x \\ T_n(x) = 2.x.T_{n-1}(x) - T_{n-2}(x) \end{cases}$$

Ces polynômes vérifient : $\langle T_i , T_j \rangle = \begin{cases} 0 \text{ si } i \neq j \\ \pi \text{ si } i = j = 0 \\ \dfrac{\pi}{2} \text{ si } i = j \neq 0 \end{cases}$

Exemple.

Supposons que l'on cherche la meilleure approximation polynômiale de degré inférieur ou égal à 2, sur $[-1 , +1]$, de $f(x) = x^3$, au sens de la norme :

$$\| f \| = \left(\int_{-1}^{+1} f^2(t)\, dt \right)^{1/2}$$

Il va tout d'abord falloir calculer les produits scalaires $< p_o , f >$, $< p_1 , f >$ et $< p_2 , f >$, où p_o , p_1 et p_2 sont les trois premiers polynômes de Legendre.

Puisque nous avons : $\begin{cases} p_0(x) = 1 \\ p_1(x) = x \\ p_2(x) = \dfrac{3x^2 - 1}{2} \end{cases}$

Il vient : $\begin{cases} < p_0 , f > = 0 \\ < p_1 , f > = \dfrac{2}{5} \\ < p_2 , f > = 0 \end{cases}$

donc le polynôme p* recherché est $p^* = \dfrac{< p_1 , f >}{< p_1 , p_1 >} p_1 = \dfrac{3}{5} x$.

Approximation polynômiale au sens de la norme de la convergence uniforme.

Les deux théorèmes suivants constituent des résultats fondamentaux. Le premier est dû à Weierstrass datant de 1885, et le deuxième à Chebyshev. Comme dans le paragraphe précédent, nous supposerons que nous désirons approximer sur un intervalle [a,b] une fonction continue f.

Théorème 6.1 (théorème d'approximation)

Pour tout $\varepsilon>0$, il existe un polynôme P tel que $\underset{[a,b]}{\text{Sup}} \, |f(x) - P(x)| \leq \varepsilon$.

Ce théorème nous montre qu'une fonction continue peut être approchée d'aussi près que possible par un polynôme. Toutefois, il ne nous donne aucune idée *a priori* du degré du polynôme à choisir en fonction de l'erreur d'approximation désirée. Un théorème dû à Berstein montre d'ailleurs que ce degré peut être aussi grand que possible.

Théorème 6.2 (théorème d'alternance de l'erreur d'approximation).

P_N^* est le polynôme de meilleure approximation uniforme de f sur [a,b] pris parmi les polynômes de degré inférieur ou égal à N si et seulement si il existe N+2 points :

$$a \leq x_0 < x_1 < x_2 < \ldots < x_{N+1} \leq b$$

vérifiant : $P_N^*(x_i) - f(x_i) = (-1)^i [P_N^*(x_0) - f(x_0)] = \pm \underset{x \in [a,b]}{\text{Sup}} \left| P_N^*(x) - f(x) \right|$

Exemple.

Cherchons à utiliser ce théorème pour trouver la meilleure approximation affine au sens de la norme de la convergence uniforme de la fonction sinus sur [0, π/2]. Soit $p_1(x) = ax + b$ cette approximation. D'après le théorème, l'erreur maximale ε est atteinte en trois points et prend des signes alternés; de plus, la concavité de la fonction sinus sur [0,π/2] implique que les deux points extrémaux où cette erreur est atteinte sont les points 0 et π/2 (voir figure 6.1).

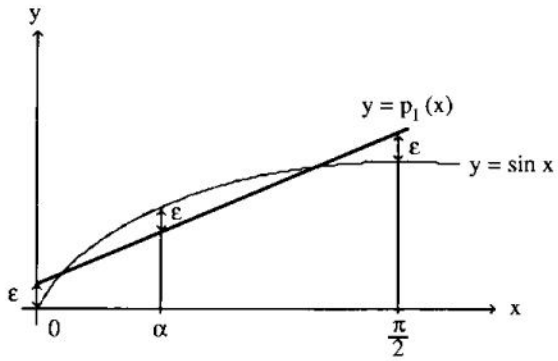


Figure 6.1

Si nous appelons α le troisième point où cette erreur est atteinte, nous avons :

$$\sin(0) - p_1(0) = -\varepsilon$$
$$\sin(\alpha) - p_1(\alpha) = +\varepsilon$$
$$\sin(\pi/2) - p_1(\pi/2) = -\varepsilon$$

de plus, l'erreur atteignant un maximum en α, il vient :

$$\sin'(\alpha) - p_1'(\alpha) = 0.$$

On en déduit sans difficulté :

$$\begin{cases} a = \dfrac{2}{\pi} \approx 0.6366 \\ b = \varepsilon = \dfrac{1}{2}\sqrt{1 - \dfrac{4}{\pi^2}} - \dfrac{1}{\pi}\operatorname{Arcos}\dfrac{2}{\pi} \approx 0.105 \\ \alpha = \operatorname{Arccos}\dfrac{2}{\pi} \end{cases}$$

On peut noter au passage que si l'on approche la fonction sinus par son développement limité à l'ordre 1 en zéro, l'erreur d'approximation est égale à $\pi/2 - 1 \approx 0.57$, elle est donc plus de cinq fois plus élevée que l'erreur commise en approximant le sinus par son polynôme de meilleure approximation uniforme de degré 1 sur $[\ 0, \pi/2]$.

Un algorithme datant de 1934, dû à Rémès ([REM34]), permet de calculer de façon itérative le meilleur approximant polynômial de degré N sur un intervalle [a,b], au sens de la norme de la convergence uniforme, d'une fonction continue f. Nous ne le donnerons pas ici car sa présentation nécessite un important développement théorique préliminaire, le lecteur qui désire de plus amples informations à son sujet peut se référer à [REM34] ou à [LAU72].

6.1.2 Approximation par des fractions rationnelles.

Le principal inconvénient des approximations polynômiales est l'existence de fonctions qui se laissent "mal" approcher, c'est-à-dire pour lesquelles, si l'on désire obtenir une précision satisfaisante, il faut utiliser un polynôme de degré élevé. Il ne faut pas croire que de telles fonctions ne sont que des "exemples d'école" : Hart & Cheney montrent dans [HCL68] que la fonction Arcsinus ne peut être approximée sur $[-1, +1]$ avec une erreur inférieure à 10^{-8} que par un polynôme de degré 10000 !!!

Pour évaluer des fonctions qui se laissent "mal" approcher par des polynômes, la seule ressource est *d'utiliser des approximations rationnelles*. On ne peut hélas plus construire de méthodes semblables à celles utilisant des polynômes orthogonaux. Toutefois le théorème suivant, dû à Chebyshev, permet de caractériser théoriquement les meilleures approximations uniformes rationnelles. Ce théorème est l'équivalent pour les fractions rationnelles du théorème d'alternance de l'erreur d'approximation rencontré auparavant.

Théorème 6.3 (Chebyshev).

R = P/Q est la meilleure approximation uniforme de f sur [a,b], prise parmi les fractions rationnelles irréductibles dont le numérateur est de degré inférieur ou égal à p et le dénominateur de degré inférieur ou égal à q, si et seulement si il existe N = 2 + Max { p + degré (P) , q + degré (Q)} points :

$$a \le x_0 < x_1 < \ldots < x_{N-1} \le b$$

$$\text{tels que : } R(x_i) - f(x_i) = (-1)^i \, (R(x_0) - f(x_0)) = \pm \sup_{x \in [a,b]} |R(x) - f(x)|$$

Il existe des algorithmes, variantes de l'algorithme de Rémès, qui permettent de construire itérativement des meilleures approximations rationnelles de fonctions

continues. On peut aussi faire appel à diverses heuristiques qui donnent de *bonnes* solutions (par exemple le calcul d'approximants de Padé [BAK75].

6.1.3 Utilisation de la méthode de Newton.

Il n'est plus besoin de présenter la méthode de Newton, bien connue des numériciens. Elle consiste, lorsque l'on désire trouver une racine α d'une fonction f continûment dérivable, à construire la suite (x_n) définie par :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

où le point de départ x_0 est choisi si possible proche de la racine recherchée.

Le théorème suivant nous prouve que si x_0 est suffisamment proche de α, alors cette suite converge vers α, et de manière *quadratique*, ce qui signifie *grosso-modo* que le nombre de chiffres significatifs sur l'estimation de α par un terme de la suite *double* à chaque pas.

Théorème 6.4 (convergence locale de la méthode de Newton)

Soit f une fonction deux fois continûment dérivable, soit α une racine *simple* de f. Il existe un voisinage V de α et un réel K strictement positif tels que si x_0 est pris dans V, alors la suite (x_n) définie par :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

reste dans V et vérifie : $|x_{n+1} - \alpha| \leq K.|x_n - \alpha|^2$.

Ce résultat nous assure une convergence uniquement *locale* de la méthode de Newton, on ne sait donc rien *a priori* sur le comportement de l'itération si le point de départ x_0 est quelconque. Il convient de faire si cela est possible une étude de convergence préliminaire chaque fois que l'on veut utiliser la méthode de Newton pour trouver les racines d'une fonction f particulière. Dans le cas où f est un polynôme, il existe cependant un résultat de convergence *globale*, énoncé par Christophe Masse dans [MAS84] et résumé par le théorème suivant.

Théorème 6.5 (Masse 1984).

Soit P un polynôme à coefficients réels dont toutes les racines sont *réelles* et *distinctes*. Notons N le degré de P, et E l'ensemble des points de départ réels x_0 pour lesquels l'itération de Newton associée à P, $x_{n+1} = x_n - P(x_n) / P'(x_n)$, diverge.

1) Si $N \leq 2$, alors E est vide ou réduit à un point.
2) Si $N = 3$, alors E est dénombrable.
3) Si $N \geq 4$, alors E est non dénombrable, et *de mesure de Lebesgue nulle.*

Ce théorème nous assure que "numériquement", l'itération de Newton associée à un polynôme dont les racines sont réelles et distinctes convergera toujours.

Application : calcul de la racine carrée.

Supposons que nous désirions évaluer la racine carrée d'un réel positif a à l'aide de la méthode de Newton. La première idée qui vient à l'esprit est de chercher la racine positive de la fonction $f(x) = x^2 - a$, ce qui conduit à construire l'itération bien connue, appelée *itération de Héron* :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Une étude sommaire ou l'utilisation du théorème 6.5 nous montrent que cette itération convergera toujours sauf si x_0 est égal à 0. La figure 6.2 nous présente graphiquement deux pas de cette itération.

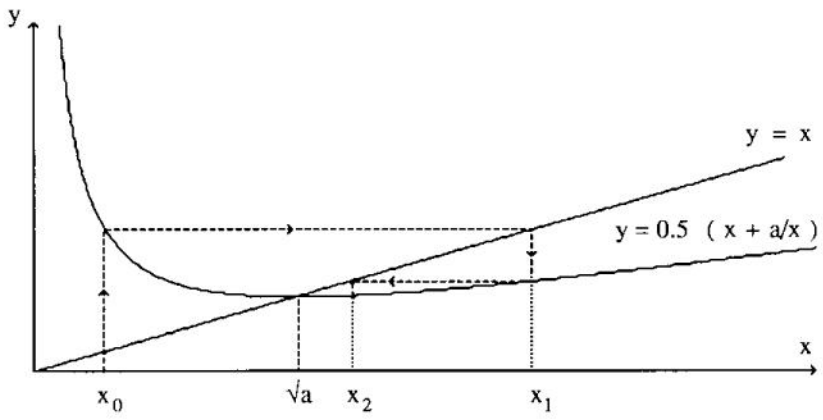


Figure 6.2 *Deux pas de l'itération de Héron.*

Si a est un nombre codé en virgule flottante normalisée sous la forme $m.B^e$ (avec |m| compris entre 1/B et 1), un moyen commode de choisir un point de départ proche de $\sqrt{a}$ (et donc de bénéficier de la convergence localement quadratique de la méthode) est de diviser l'exposant de a par 2, ce qui conduit à prendre par exemple comme point de départ $B^{(e \text{ div } 2)+1}$, où "div" représente la division euclidienne. Des stratégies plus fines de choix d'un bon point de départ pour l'itération de Héron sont présentées dans [MAS84].

Si on utilise une machine qui effectue les divisions nettement plus lentement que les multiplications, la division qui apparait dans l'itération de Héron peut être pénalisante. On peut alors avoir avantage à utiliser une autre itération de Newton, qui permet de calculer $1/\sqrt{a}$ (on retrouve ensuite $\sqrt{a}$ non pas en inversant ce nombre — on

veut éviter les divisions ! — mais en le multipliant par a). Cette itération s'obtient en cherchant par la méthode de Newton la racine positive de $f(x) = 1/x^2 - a$, ce qui nous donne :

$$x_{n+1} = \frac{1}{2} x_n \left(3 - a x_n^2\right)$$

La figure 6.3 présente deux pas de cette itération.

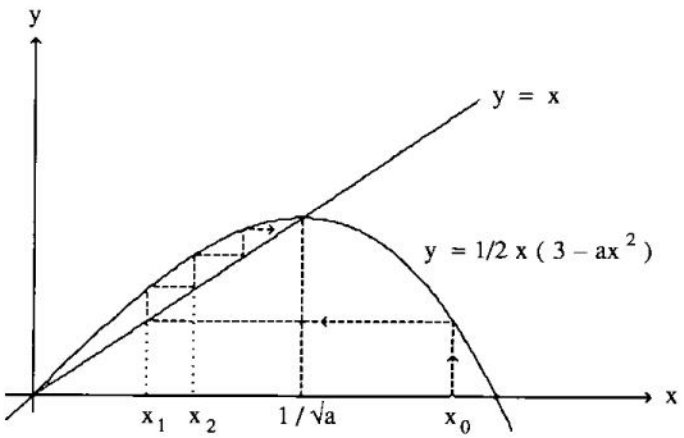


Figure 6.3 *Deux pas de l'itération associée au calcul de $1/\sqrt{a}$.*

Cette fois ci, la convergence de l'itération n'est plus globale. Une étude élémentaire permet de montrer qu'il y a convergence pour tout x_0 pris dans $I =]\,0, \sqrt{3}/\sqrt{a}\,[$. Si le nombre a est codé en virgule flottante sous la forme $m.B^e$ (avec $1/B \leq |m| < 1$), alors il est facile de constater que $B^{-(e \text{ div } 2) - 1}$ ("div" désigne toujours la division euclidienne) appartient forcément à I. C'est donc un point de départ tout indiqué.

6.2 CALCUL DES FONCTIONS ELEMENTAIRES PAR ADDITIONS ET DECALAGES.

Nous allons maintenant étudier des méthodes de calcul des fonctions élémentaires permettant d'éviter les multiplications et les divisions. Ces méthodes sont particulièrement adaptées au calcul de fonctions *par matériel.*

Ce domaine a été abordé en 1959 par J. Volder ([VOL59]), qui présenta l'algorithme *CORDIC* (COordinate Rotations on a DIgital Computer), qui permet de calculer les fonctions sinus, cosinus et arctangente, et ce en n'effectuant que des additions, des soustractions, et des décalages (en base 2) .

En 1971, J. Walther montre dans [WAL71] que CORDIC, moyennant de petites modifications, peut être étendu au calcul des fonctions hyperboliques, ainsi qu'à l'exponentielle, au logarithme et à la racine carrée. Il constate en outre que des algorithmes usuels de multiplication et de division (non restaurante) peuvent s'exprimer comme un cas particulier de CORDIC (il est d'ailleurs à noter que dès 1962, J.E. Meggitt était à même de calculer certaines fonctions élémentaires par des procédés de "pseudo-division" et de "pseudo-multiplication"). On obtient ainsi un *schéma unifié* permettant de calculer les principales fonctions mathématiques, et qui se prête particulièrement bien à une réalisation matérielle (CORDIC a d'ailleurs été choisi pour la calculatrice HP 35 de Hewlett-Packard [SB73]).

L'algorithme CORDIC, dans la version de Walther, est basé sur l'itération à trois variables suivante :

$$x_{n+1} = x_n - m d_n y_n 2^{-\sigma(n)}$$
$$y_{n+1} = y_n + d_n x_n 2^{-\sigma(n)}$$
$$z_{n+1} = z_n - d_n e_{\sigma(n)}$$

où les choix du paramètre m, des variables d_n et des constantes e_n et de la fonction sont présentés figure 6.4. Sur cette figure, dans les cas notés * (fonctions trigonométriques et arithmétiques), $\sigma(n)$ est égal à n, et dans les cas notés ** (fonctions hyperboliques), $\sigma(n)$ est égal à :

n si $1 \leq n \leq 4$
$n-1$ si $5 \leq n \leq 14$
$n-2$ si $15 \leq n \leq 42$
$n-3$ si $43 \leq n \leq 124$ ...

De manière générale, dans le cas des fonctions hyperboliques, $\sigma(n)$ est égal à $n-p+1$ si $u_p \leq n \leq u_{p+1}-1$, où les termes u_p sont donnés par :

$u_1 = 1$
$u_{p+1} = 3u_p + 4 - 2p.$

Dans une implantation pratique, lorsqu'on calcule les fonctions hyperboliques, on fait comme si $\sigma(n)$ était égal à n, et on répète les itérations d'indices 4, 13, 40, 121... (si on répète l'itération k, on répètera l'itération 3k+1).

Les constantes K et K' valent respectivement :

$$\prod_{i=0}^{\infty} \cos e_i = \left(\prod_{i=0}^{\infty} (1 + 2^{-2i}) \right)^{-1/2} \quad \text{et} \quad \prod_{i=1}^{\infty} \mathrm{ch}\, e_{\sigma(i)} = \left(\prod_{i=1}^{\infty} (1 - 2^{-2\sigma(i)}) \right)^{-1/2}$$

fonction	valeurs init.	m	e_n	d_n	résultats
sin, cos*	$x_0 = K$ $y_0 = 0$	1	$\text{arctg}\,2^{-n}$	$\text{sign}\ z_n$	$\begin{cases} x_n \longrightarrow \cos z_0 \\ y_n \longrightarrow \sin z_0 \end{cases}$
arctg*	$z_0 = 0$	1	$\text{arctg}\,2^{-n}$	$-\text{sign}\ y_n$	$z_n \longrightarrow \text{arctg}\,\frac{y_0}{x_0}$
sh, ch**	$x_1 = K'$ $y_1 = 0$	-1	$\text{argth}\,2^{-n}$	$\text{sign}\ z_n$	$\begin{cases} x_n \longrightarrow \text{ch}\, z_0 \\ y_n \longrightarrow \text{sh}\, z_0 \end{cases}$
argth**	$z_1 = 0$	-1	$\text{argth}\,2^{-n}$	$-\text{sign}\ y_n$	$z_n \longrightarrow \text{argth}\,\frac{y_0}{x_0}$
multiplica* tion	$y_0 = 0$	0	2^{-n}	$\text{sign}\ z_n$	$y_n \longrightarrow x_0 z_0$
division*	$z_0 = 0$	0	2^{-n}	$-\text{sign}\ y_n$	$z_n \longrightarrow \frac{y_0}{x_0}$

Figure 6.4 *L'algorithme CORDIC dans la version de J. Walther.*

L'exponentielle est obtenue par addition des fonctions sh et ch, et le logarithme est obtenu en effectuant un changement de variable permettant d'utiliser la relation :

$$\text{argth}\ x = \frac{1}{2}\,\text{Log}\left|\frac{1+x}{1-x}\right|.$$

En 1970, De Lugish [LUG70] étudie une classe d'algorithmes similaires, tandis que Chen [CHE72] développe des idées originales à ce sujet. Nous allons dégager les principes communs à tous ces algorithmes, en présentant un nouveau concept : celui de *base discrète*, qui nous permettra de mettre au point de nouvelles méthodes et de montrer l'étroite parenté qui lie certains algorithmes déjà connus.

Tout d'abord, nous allons analyser la petite fonction suivante, écrite en pseudo-PASCAL, qui correspond en base 2 à un algorithme inventé par Briggs il y a environ trois siècles, et qui n'utilise que des décalages et des additions, pour peu que l'on travaille en base 2. N est un paramètre dont nous allons étudier l'influence, et on suppose que les constantes Log $(1+2^{-i})$ (pour i allant de zéro à N) sont calculées à l'avance et mémorisées en mémoire morte.

Les figures 6.5 à 6.8 présentent le graphe de cette fonction entre 0 et 2 pour différentes valeurs de N. Qu'observons-nous ? Lorsque N tend vers l'infini, si t est compris entre 0 et une borne qu'il reste à déterminer, la valeur retournée par cette fonction semble converger vers l'exponentielle de t. De même, on peut constater expérimentalement que la valeur finale de la variable "x" de l'algorithme semble converger vers t (ces remarques sont liées, car "ex = e^x" est trivialement un invariant de la boucle "**pour i allant de** 0 **jusqu'à** N **faire**" de l'algorithme).

Bien sûr ces résultats doivent être démontrés, ce qui sera fait par la suite, mais admettons-les momentanément.

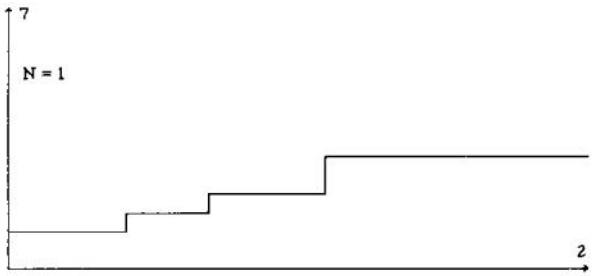


Figure 6.5

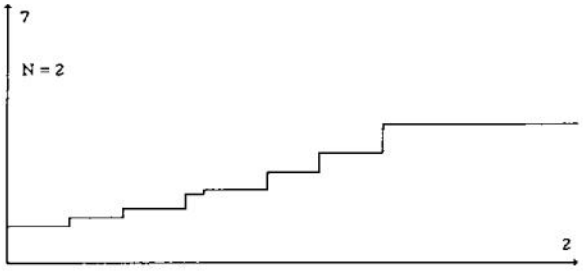


Figure 6.6

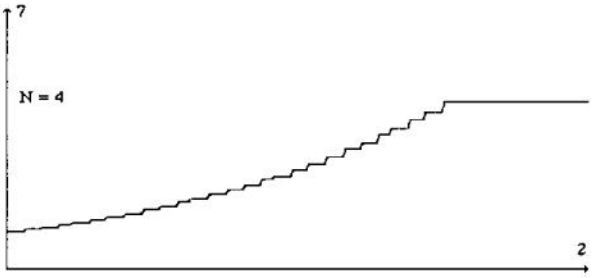


Figure 6.7

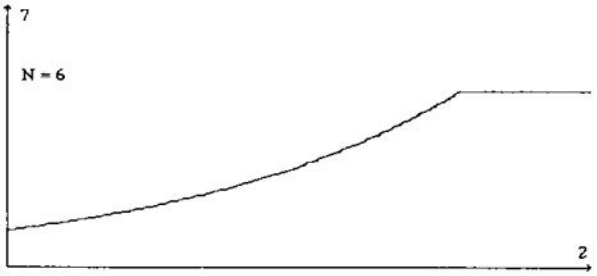


Figure 6.8

```
Fonction expo (t : réel) : réel ;
var i : entier ; x, ex, u : réel ;
début
    x := 0; ex := 1;
    pour i allant de 0 jusqu'à N faire
    début
        u := x + Log (1 + 2^-i);
        si u ≤ t alors
        début
            x := u;
            ex := ex + ex.2^-i
        fin
    fin
    expo := ex;
fin.
```

Définissons une suite (d_i) par :

$$d_i = \begin{cases} 1 \text{ si le test "u≤t" s'est avéré vrai à la boucle d'indice i de l'algorithme} \\ 0 \text{ sinon} \end{cases}$$

il est clair que la valeur x_i de la variable x de l'algorithme après exécution de la boucle d'indice i vaut :

$$x_i = \sum_{k=0}^{i} d_k \text{Log}\,(1+2^{-k}).$$

Et donc il semble d'après les résultats que nous avons admis que l'on ait :

$$t = \sum_{k=0}^{\infty} d_k \text{Log}\,(1+2^{-k})$$

Ce principe d'écriture d'un nombre réel sous la forme $\Sigma\, d_k e_k$, où les e_k sont des constantes précalculées et les d_k des valeurs prises dans {0,1}, est à la base d'une large classe d'algorithmes de calcul de fonctions mathématiques usuelles. C'est ce principe que nous allons étudier à présent, en essayant de l'étendre à d'autres valeurs des d_k, dans le but de pouvoir calculer sur des machines qui travaillent dans des bases différentes de 2. Nous aborderons deux aspects :

– La recherche des suites (e_n) telles que tout élément x d'un domaine donné (en pratique un intervalle) puisse s'écrire sous la forme :

$$x = d_0e_0 + \ldots + d_ne_n + \ldots$$

où les d_i seront compris entre 0 et un entier naturel p . De telles suites seront appelées *Bases discrètes additives*, nous allons les étudier maintenant.

– La connaissance d'algorithmes permettant de calculer les termes d_i correspondant à un x donné, puis l'utilisation de ces algorithmes au calcul de certaines fonctions.

6.2.1 bases discrètes additives.

Nous noterons S l'ensemble des suites réelles, strictement positives, décroissantes et sommables.

Définition 6.1.

Soit $E = (e_n)$ un élément de S, on appellera *ensemble généré d'ordre p de E* l'ensemble :

$$G_p(E) = \left\{ \sum_{n=0}^{\infty} d_ne_n \;\middle|\; d_n \in \{0, 1, 2, \ldots, p\} \right\}.$$

$G_p(E)$ représente l'ensemble des nombres que l'on peut représenter sous la forme $d_0e_0 + \ldots + d_ne_n + \ldots$

Le cas qui nous intéresse dans cette étude est celui où cet ensemble est un *intervalle*, car cela nous permettra ultérieurement de calculer des fonctions sur cet intervalle. D'où la définition qui suit :

Définition 6.2.

Soit E un élément de S. On dira que E est une *base discrète additive d'ordre p* si $G_p(E)$ est un intervalle.

N.B. Par abus de notation, on appelera les bases discrètes additives "bases discrètes", sauf lorsqu'il y aura risque de confusion avec les bases discrètes *multiplicatives*, que nous allons bientôt étudier.

On peut d'ores et déjà remarquer que nous utilisons tout les jours des bases discrètes, en écrivant les nombres en notation décimale. En effet, dire qu'un réel x compris entre 0 et 10 peut s'écrire sous la forme : $x = a_0.a_1a_2a_3a_4\ldots$ (a_i est la ième décimale de x) revient à dire que x est égal à :

$$\sum_{i=0}^{\infty} a_n\, 10^{-n}.$$

On utilise donc la base discrète d'ordre 9 : $(10^{-n})_{n\in\mathbb{N}}$.

En un certain sens la notion de base discrète généralise donc celle de *base de numération*. Le théorème suivant, qui permet de caractériser les bases discrètes, nous permettra d'ailleurs d'entrevoir cette généralisation.

Théorème 6.6.

$E = (e_n) \in S$ est une base discrète d'ordre p si seulement si pour tout entier n :

$$e_n \leq p \sum_{k=n+1}^{\infty} e_k \qquad (A) .$$

Démonstration.

a. La condition (A) est nécessaire.

Soit $E = (e_n)$ un élément de S, supposons qu'il existe un entier naturel n tel que :

$$e_n > p \sum_{k=n+1}^{\infty} e_k = r_n$$

et posons $I =]r_n , e_n[$. montrons que l'intersection de I avec $G_p(E)$ est vide.

En effet, soit $a \in G_p(E)$, soit (d_n) la suite telle que $a = \sum_{i=0}^{\infty} d_i e_i$.

1) S'il existe $k \leq n$ tel que $d_k \neq 0$, alors $a \geq d_k e_k \geq e_k \geq e_n$ puisque la suite (e_i) décroit.

2) Si pour tout $k \leq n$, $d_k = 0$, alors $a \leq p \sum_{k=n+1}^{\infty} e_k$.

Par conséquent, dans tous les cas, $a \notin I$.

b. La condition (A) est suffisante.

Pour établir ceci, montrons que si la condition (A) est vérifiée, alors, pour tout réel a appartenant à l'intervalle

$$I = \left[0 , p \sum_{n=0}^{\infty} e_n \right]$$

la suite (d_n) définie par

$$a_0 = 0$$
$$d_i = \text{Max} \{ j \in \{0, 1, 2, \ldots, p\} \mid a_i + j e_i \leq a \}$$
$$a_{i+1} = a_i + d_i e_i$$

nous assure : $\sum_{n=0}^{\infty} d_n e_n = a$.

Notre problème revient à démontrer que a est la limite de la suite a_n ; dans ce but, établissons par récurrence l'inégalité :

$$| a_n - a | \leq p \sum_{k=n}^{\infty} e_k .$$

– Elle est vérifiée si n = 0 puisque a ∈ I.

– Supposons qu'elle soit vraie pour n ≥ 0, et évaluons la quantité $| a_{n+1} - a |$.

1) Si $d_n < p$, alors, puisque d_n est égal à Max $\{j \in \{0, \ldots, p\} / a_n + je_n \leq a\}$ nous avons :

$$a_{n+1} = a_n + d_n e_n \leq a \leq a_n + (d_n + 1)\, e_n$$

et par conséquent : $a_{n+1} \leq a \leq a_{n+1} + e_n$; soit, en utilisant (A) :

$$| a_{n+1} - a | \leq p \sum_{k=n+1}^{\infty} e_k$$

2) Si $d_n = p$, alors $a_{n+1} = a_n + pe_n \leq a$, donc d'après l'hypothèse de récurrence :

$$|a_{n+1} - a| = |a - a_n| - pe_n \leq \left(p \sum_{k=n}^{\infty} e_k \right) - p\, e_n$$

$$\leq p \sum_{k=n+1}^{\infty} e_k$$

Le théorème est donc démontré.

Ce théorème permet de prouver facilement que la suite (a^{-n}) est une base discrète d'ordre p si et seulement si $1 < a \leq p+1$. Pour les valeurs entières de a nous retrouvons les bases usuelles de numération (la suite (10^{-n}) est une base discrète d'ordre 9), mais nous pouvons par exemple dire que *la suite (π^{-n}) est une base discrète d'ordre 3.*

On peut donc par exemple écrire "en base π" 1/2 sous la forme "0.11211202…", ce qui signifie qu'il existe une suite (d_n), $d_n \in \{0,1,2,3\}$, telle que :

$$\frac{1}{2} = \sum_{n=0}^{\infty} d_n \pi^{-n}$$

avec $d_0 = 0$, $d_1 = d_2 = 1$, $d_3 = 2$, etc.

Les bases discrètes de la forme (a^{-n}) seront appelées *bases discrètes géométriques*. Nous étudierons ultérieurement des algorithmes permettant de calculer les termes d_i. Auparavant, un second théorème nous permettra de prouver plus aisément que certaines suites sont des bases discrètes.

Théorème 6.7.

Si $E = (e_n) \in S$ est une base discrète d'ordre p et si f est une fonction à variable réelle vérifiant :

1) f ' est continue.
2) f(0) = 0
3) f est *concave* et *strictement croissante.*

alors la suite $(f(e_n))$ est une base discrète d'ordre p.

Ce théorème permet de prouver, et ceci sera fondamental dans la suite de cette étude, que les suites $e_n = Arctg\ (a^{-n})$ et $e'_n = Log\ (1 + a^{-n})$ sont des bases discrètes d'ordre p lorsque $1 < a \leq p+1$. Ceci nous permet, par exemple, d'écrire 2 en base $Log\ (1 + \pi^{-n})$ sous la forme "2.2012202...".

Démonstration du théorème.

a) $(f(e_n))$ est une suite décroissante de réels strictement positifs.

Ceci provient de la croissance stricte de f. pour tout entier naturel n, $e_n \geq e_{n+1} > 0$, donc $f(e_n) \geq f(e_{n+1}) > f(0) = 0$.

b) la série $\sum_{n=0}^{\infty} f(e_n)$ *converge.*

Pour montrer ceci, il suffit d'établir que pour tout entier naturel n, $f(e_n)$ est inférieur ou égal à $e_n f'(0)$.

En effet, f est concave et f ' est continue sur I, donc f ' est décroissante sur I. par conséquent :

$$f(e_n) = \int_0^{e_n} f'(x)\ dx \leq f'(0)\, e_n$$

c) La suite $(f(e_n))$ vérifie la condition (A) du théorème 6.6.

La concavité de f sur (I) implique que pour tout $(a_1, a_2, \ldots, a_n) \in I^n$ $(a_i \geq 0)$, tel que $p(a_1 + a_2 + \ldots + a_n)$ est inclus dans I, nous avons :

$$f[p(a_1 + \ldots + a_n)] \leq p\,[f(a_1) + \ldots + f(a_n)]$$

Par conséquent, f ' étant continue, si la série $p\sum a_i$ converge vers un élément de I, alors :

$$f\left(p\sum_{i=0}^{\infty} a_i\right) \leq p\left(\sum_{i=0}^{\infty} f(a_i)\right)$$

Or, pour tout entier naturel n, nous avons $e_n \leq p\sum_{i=n+1}^{\infty} e_i$

par conséquent $f(e_n) \leq f\left(p\sum_{i=n+1}^{\infty} e_i\right) \leq p\sum_{i=n+1}^{\infty} f(e_i)$

Ce qu'il fallait démontrer.

Les figures 6.9 à 6.11 donnent des exemples d'ensembles G_p (E) pour différentes suites E.

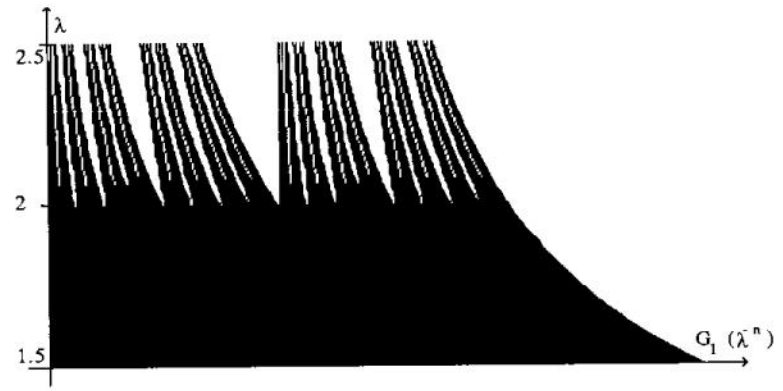


Figure 6.9. *Ensembles* G_1 (λ^{-n}), λ *variant continûment entre 1.5 et 2.5*

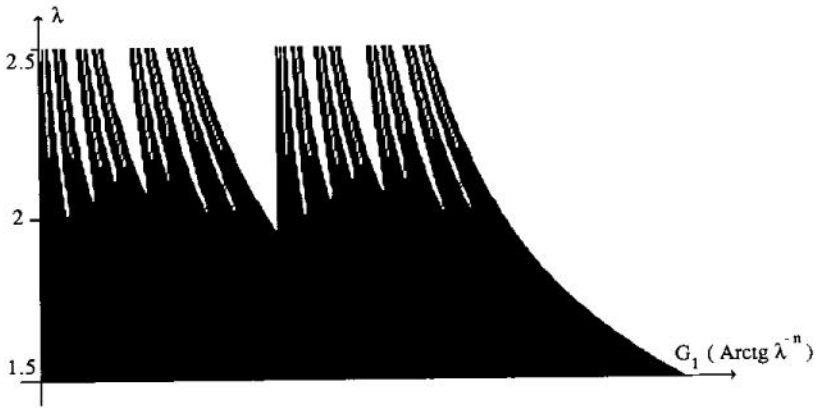


Figure 6.10. *Ensembles* G_1 (Arctg λ^{-n}), λ *variant continûment entre 1.5 et 2.5*

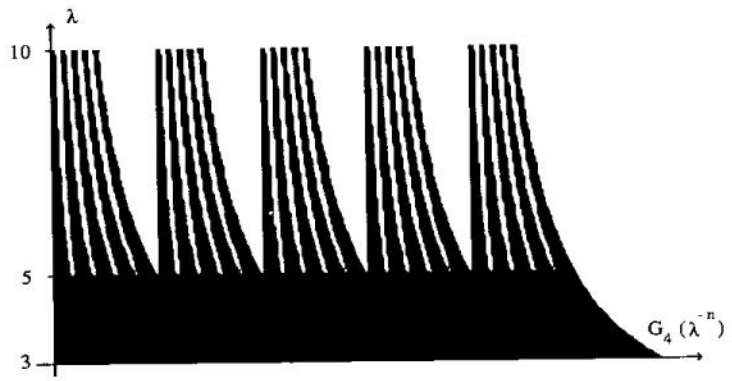


Figure 6.11. *Ensembles* $G_4(\lambda^{-n})$, λ *variant continûment entre 3 et 10*

Exemple d'application.

Les algorithmes de calcul des coefficients d_i seront explicités plus tard, mais on peut d'ores et déjà constater que si l'on dispose d'une machine travaillant en base 2, étant donné que la suite $e_n = \mathrm{Log}\,(1 + 2^{-n})$ est une base discrète d'ordre 1, on peut écrire tout élément x de l'intervalle $[0\ ,\ \Sigma ei]$ sous la forme :

$$x = \sum_{i=0}^{\infty} d_i \,\mathrm{Log}\,(1+2^{-i})\ ,\ d_i \in \{0,1\}$$

et donc obtenir son exponentielle :

$$e^x = \prod_{i=0}^{\infty} (1+2^{-i})^{d_i}\ .$$

Ce produit infini (tronqué bien entendu à un rang dépendant de la précision désirée) peut être calculé sans multiplications puisque :

1) un produit par une constante de la forme $(1+2^{-i})$ s'effectue au moyen d'un décalage et d'une addition.

2) les "exposants" d_i valant 0 ou 1 aucun calcul de puissance n'est nécessaire.

Cet algorithme de calcul de l'exponentielle sera explicité plus longuement ultérieurement. Auparavant, il va falloir introduire une autre notion : celle de bases discrètes *multiplicatives*. En effet, nous venons d'étudier un moyen de représenter les nombres réels comme *sommes* de constantes prédéfinies; pour calculer certaines fonctions (logarithme, racine carrée...) il faudra écrire les nombres comme *produits* de constantes. D'où les définitions suivantes.

6.2.2 bases discrètes multiplicatives.

Définition 6.3.

Notons M l'ensemble des suites réelles (e_n) dont les termes sont décroissants, strictement supérieurs à 1, et dont le produit infini converge. On appellera *ensemble généré multiplicatif d'ordre p* de $E = (e_n) \in M$ l'ensemble :

$$GM_p(E) = \left\{ \prod_{n=0}^{\infty} e_n^{d_n} \;\middle|\; d_n \in \{0, 1, 2, \ldots, p\} \right\}$$

Définition 6.4.

On dira que $E \in M$ est une *base discrète multiplicative d'ordre p* si $GM_p(E)$ est un intervalle.

Il ne sera pas nécessaire de chercher des résultats semblables aux théorèmes 6.6 et 6.7 pour caractériser les bases discrètes multiplicatives, on peut en effet se ramener au cas des bases additives grâce au résultat suivant, dont la démonstration est évidente.

Théorème 6.8.

La suite $E = (e_n) \in M$ est une base discrète multiplicative d'ordre p si et seulement si la suite $(\mathrm{Log}\,(e_n))$ est une base discrète additive d'ordre p.

Exemple d'application.

Les théorèmes 6.7 et 6.8 permettent de montrer facilement que la suite $e_n = ((1 + 2^{-n})^2)$ est une base discrète multiplicative d'ordre 1. On peut donc, par le biais d'un algorithme que nous étudierons ultérieurement, écrire tout nombre x de l'intervalle $[1, \prod e_i]$ sous la forme :

$$x = \prod_{n=0}^{\infty} \left((1 + 2^{-n})^2 \right)^{d_n}$$

et obtenir immédiatement sa racine carrée :

$$\sqrt{x} = \prod_{n=0}^{\infty} (1 + 2^{-n})^{d_n}$$

Le produit infini est tronqué à un rang dépendant de la précision désirée. Là encore, ce produit est très simple à calculer car une multiplication par un terme de la forme $(1 + 2^{-n})$ s'effectue au moyen d'un décalage et d'une addition.

6.2.3 Algorithmes de calcul des coefficients d_i.

Nous examinerons tout d'abord des algorithmes de décomposition d'un nombre sur une base discrète additive. Le cas des bases discrètes multiplicatives s'en déduira immédiatement par analogie. Supposons donc que nous désirions trouver la décomposition $\sum d_n e_n$ correspondant à un nombre réel t. Le principe des deux algorithmes que nous allons étudier est de construire terme à terme une suite t_i qui converge vers t, et telle que :

$$t_i = \sum_{n=0}^{i} d_n e_n \ .$$

Pour ceci, nous emploierons une stratégie de type "algorithme glouton", c'est-à-dire que pour passer de l'étape i à l'étape i+1, nous choisirons la valeur de d_{i+1} qui rend t_{i+1} *le plus proche possible* de t. Nous envisagerons ici 2 cas :

– On désire des valeurs de d_i comprises entre 0 et un entier naturel p : l'algorithme que nous utiliserons s'appelera algorithme *unidirectionnel* (ce nom provient du fait que la suite t_i croît vers t).

– On désire des valeurs de d_i comprises entre -p et p : nous utiliserons un algorithme appelé algorithme *bidirectionnel* (la suite t_i "oscille" autour de t).

Ces deux "algorithmes gloutons" sont présentés par les théorèmes suivants.

Théorème 6.9. (Algorithme unidirectionnel).
Si $E = (e_n)$ est une base discrète additive d'ordre p, alors pour tout élément de $[0 , p\sum e_i]$, les suites (t_n) et (d_n) définies par :

$$t_0 = 0$$
$$d_n = \text{Max} \{ j \le p \mid t_n + j e_n \le t \}$$
$$t_{n+1} = t_n + d_n e_n \ .$$

vérifient : $t = \lim_{n \to \infty} t_n = \sum_{n=0}^{\infty} d_n e_n$.

Nous ne démontrerons pas ce théorème 6.9, car il n'est autre que la condition suffisante du théorème 6.6, nous l'avons replacé ici pour plus de clarté. Il est à noter que lorsqu'on utilise une base discrète d'ordre 1 (ce qui sera toujours le cas pour les algorithmes implantés sur une machine binaire), l'algorithme de choix de d_n se simplifie considérablement et devient : $d_n = 1$ *si* $t_n + e_n \le t$; $d_n = 0$ *sinon.*

Théorème 6.10. (Algorithme bidirectionnel).

Si $E = (e_n)$ est une base discrète additive d'ordre p, alors pour tout élément de $[-p\Sigma\, e_i\ ,\ +p\Sigma\, e_i]$, les suites (t_n) et (d_n) définies par :

$$t_0 = 0$$
$$\textbf{Si } t_n \le t \textbf{ alors } d_n = \text{Max}\ \{1 \le j \le p \mid t_n + (j-1)\, e_n \le t\}$$
$$\textbf{sinon } d_n = \text{min}\ \{-p \le j \le -1 \mid t_n + (j+1)\, e_n \ge t\}$$
$$t_{n+1} = t_n + d_n e_n\ .$$

vérifient : $t = \lim_{n \to \infty} t_n = \sum_{n=0}^{\infty} d_n e_n\ .$

La démonstration de ce résultat est semblable à celle de la condition suffisante du théorème 6.6.

Là encore, lorsqu'on utilise une base discrète d'ordre 1, l'algorithme de choix de d_n se simplifie considérablement et devient : $d_n = -1$ *si* $t_n > t$; $d_n = 1$ *sinon.*

On peut de la même manière exhiber des algorithmes multiplicatifs bidirectionnel et unidirectionnel. Je ne connais pas d'utilisation pratique de l'algorithme multiplicatif bidirectionnel, mais l'algorithme multiplicatif unidirectionnel (qui se déduit immédiatement de l'algorithme additif unidirectionnel par analogie) est donné par le théorème suivant.

Théorème 6.11. (Algorithme unidirectionnel multiplicatif).

Si $E = (e_n)$ est une base discrète multiplicative d'ordre p, alors pour tout élément de $[1\ ,\ (\prod e_i)]$, les suites (t_n) et (d_n) définies par :

$$t_0 = 1$$
$$d_n = \text{Max}\ \{j \le p \mid t_n \,.\, e_n^{\,j} \le t\}$$
$$t_{n+1} = t_n \,.\, e_n^{\,d_n}\ .$$

vérifient : $t = \lim_{n \to \infty} t_n = \prod_{n=0}^{\infty} e_n^{d_n}\ .$

Tout comme dans le cas de l'algorithme unidirectionnel additif, le choix de d_n est simplifié lorsque (e_n) est une base discrète multiplicative d'ordre 1, et devient :

$$d_n = 1 \textit{ si } t_n \,.\, e_n \le t\ ;\ \ d_n = 0 \textit{ sinon.}$$

6.2.4 Calcul de l'exponentielle et du logarithme.

Calcul de l'exponentielle.

Nous allons ici étudier deux applications des algorithmes que nous venons de présenter : le calcul de la fonction exponentielle à l'aide de l'algorithme unidirectionnel *additif*, et le calcul du logarithme à l'aide de l'algorithme unidirectionnel *multiplicatif*.

Nous avons vu précédemment que si a est un réel vérifiant $1 < a \leq p+1$ $(p \in \mathbb{N})$, alors la suite $(\text{Log}\ (1 + a^{-n}))$ est une *base discrète additive d'ordre p*. L'idée de base de l'algorithme que nous étudions ici est de décomposer un nombre réel x sous la forme :

$$(1) \qquad x = \sum_{n=0}^{\infty} d_n \text{Log}\ (1 + a^{-n}) \qquad d_n \in \{0, 1, \ldots, p\}$$

en utilisant l'algorithme unidirectionnel, afin d'obtenir comme résultat :

$$(2) \qquad e^x = \prod_{n=0}^{\infty} (1 + a^{-n})^{d_n}$$

Or, on sait que sur une machine travaillant en base B, les multiplications par une puissance de B peuvent s'éxécuter d'une manière très simple, à l'aide d'un *décalage*. Ici, on aura avantage à choisir a = B, et donc à utiliser la base discrète additive d'ordre B-1 $(\text{Log}\ (1 + B^{-n}))$, car une multiplication par $(1 + B^{-n})$ se ramènera à une addition et une multiplication par B^{-n}. Il est évident que la série (1) et le produit infini (2) seront tronqués à un rang N dépendant de la précision ε désirée. Nous présentons tout d'abord l'algorithme obtenu, et nous expliciterons ensuite la relation entre N et ε. Cet algorithme, comme celui qui suivra, nécessite la mémorisation des constantes $\text{Log}\ (1 + B^{-n})$. Il n'est pas inutile de préciser que, étant donné qu'à la précision machine près, $\text{Log}\ (1 + B^{-n})$ devient très vite égal à B^{-n}, il ne sera nécessaire de mémoriser qu'un petit nombre de constantes.

Exponentielle.

(* Cet algorithme calcule e^t. Il converge pour $0 \le t \le (B-1)\sum_{i=0}^{\infty} \mathrm{Log}\,(1 + B^{-i})$.

Précision relative : environ B^{-n}, résultat : la valeur finale de la variable *exp* *)

```
Début
    x := 0 ;
    exp := 1 ;
    pour k := 0 jusqu'à N faire
      début
        d := 0 ;
        u := 0 ;
        tant que (u < t) et (d < B - 1) faire
          début
            u := x + Log (1 + B^-k) ;
            si u ≤ t alors
              début
                x := u ;
                exp := exp + exp.B^-k
              fin ;
            d := d + 1
          fin
      fin
Fin.
```

Il peut être intéressant de noter qu'en base 2, cet algorithme se simplifie et devient l'algorithme *expo* que nous avons analysé au début du paragraphe 6.2.

Evaluation du nombre N de pas nécessaires.

Lorsque nous approchons une variable t à l'aide d'une suite (t_n) générée par l'algorithme unidirectionnel de décomposition sur une base discrète d'ordre p, (e_n), l'erreur au $n^{ième}$ pas, $|t_n - t|$, est majorée par :

$$p \sum_{i=n+1}^{\infty} e_i = r_n \ .$$

Dans le cas qui nous concerne, $e_i = \mathrm{Log}\,(1 + B^{-i})$ est strictement inférieur à B^{-i}, par conséquent le terme r_n est majoré par

$$(B\text{-}1) \sum_{i=n+1}^{\infty} B^{-i} = B^{-n} .$$

On en déduit que l'erreur relative sur le résultat "exp" de l'algorithme précédent, égale à $| e^{t_n} - e^{t} | / e^{t}$, sera majorée par $e^{B^{-n}} - 1 \approx B^{-n}$, ce qui en gros nous assure que si l'on désire calculer la fonction exponentielle avec N chiffres significatifs, il faudra effectuer N pas de l'algorithme. Il est à noter que si α est un réel strictement supérieur à 1, et si dans l'algorithme précédent on remplace les constantes $(\mathrm{Log}\,(1+B^{-n}))$ par les constantes $(\mathrm{Log}_{\alpha}(1+B^{-n}))$, nous obtiendrons un algorithme de calcul de l'exponentielle en base α.

Calcul du logarithme.

Sur une machine travaillant en base B, l'idée fondamentale de cet algorithme est de décomposer un nombre réel x sur la base discrète multiplicative d'ordre B-1 $(1 + B^{-n})$:

$$x = \prod_{n=0}^{\infty} (1 + B^{-n})^{d_n} \qquad d_n \in \{0, 1, \ldots, B\text{-}1\}$$

en utilisant l'algorithme unidirectionnel multiplicatif, afin d'obtenir :

$$\mathrm{Log}\, x = \sum_{n=0}^{\infty} d_n \mathrm{Log}\left(1 + B^{-n}\right)$$

Cet algorithme de calcul du logarithme apparaît ainsi comme une sorte d'algorithme "dual" de l'algorithme de calcul de l'exponentielle présenté auparavant. Cette notion de dualité d'algorithmes peut d'ailleurs revêtir un sens plus profond, présenté dans [MUL85], où l'on montre que si l'on sait calculer une fonction monotone f à l'aide de l'algorithme bidirectionnel où de l'algorithme unidirectionnel, alors on peut construire très facilement un algorithme de calcul de f^{-1}. Cet algorithme de calcul du logarithme est présenté ci-dessous.

Logarithme.

(* calcule Log t, $t \in \left[1 \ , \left(\prod_{i=0}^{\infty} (1 + B^{-i}) \right)^{(B-1)} \right]$, résultat : la valeur finale de L *)

```
Début
  x := 1 ;
  L := 0 ;
  pour k := 0 jusqu'à N faire
    début
      d := 0 ;
      u := 0 ;
      tant que (u < t) et (d < B-1) faire
         début
           u := x + x.B^-k ;
           si (u ≤ t) alors
              début
                x := u ;
                L := L + Log (1 + B^-k)
              fin ;
           d := d + 1 ;
         fin
    fin
Fin.
```

Tout comme dans le cas de l'exponentielle, cet algorithme se simplifie lorsque B est égal à 2, et devient :

```
Début
    x := 1 ; L := 0 ;
    pour  k := 0 jusqu'à N faire
        début
            u := x + x.2^-k
            si u ≤ t alors
                début
                    x := u ;
                    L := L + Log (1 + 2^-k)
                fin
        fin
Fin.
```

Nombre N de pas nécessaires.

On peut montrer, de même manière que précedemment, qu'après exécution de l'algorithme, $| L - \text{Log}(t) | \leq B^{-N}$.

Remarque. De même que pour l'algorithme précédent, si l'on remplace les constantes (Log $(1+B^{-n})$) par les constantes ($Log_{\alpha}(1+B^{-n})$), nous obtiendrons un algorithme de calcul de l'exponentielle en base α.

6.2.5 Calcul de l'exponentielle complexe.

Nous allons maintenant utiliser les algorithmes unidirectionnel et bidirectionnel pour élaborer un algorithme permettant de calculer l'exponentielle complexe dans un domaine que nous expliciterons ultérieurement. Par souci de simplification, nous nous cantonnerons au calcul sur une machine binaire, bien qu'une généralisation à d'autres bases de numération soit tout à fait envisageable. Nous retrouverons comme cas particuliers de cet algorithme ceux de calcul de l'exponentielle et du logarithme réels exposés précedemment, ainsi que l'algorithme CORDIC de calcul des fonctions trigonométriques introduit par Jack Volder en 1959 .

Les équations

Nous désirons calculer les nombres réels a et b tels que $a + ib = e^{x+iy}$, où x et y sont donnés, en utilisant la relation bien connue : $e^{x+iy} = (\cos y + i.\sin y) . e^{x}$. Par conséquent, si nous désirons décomposer x et y sur des bases discrètes additives facilitant le calcul, nous aurons à satisfaire deux exigences :

– La base discrète (e_n^x) associée à la variable x doit être choisie de sorte qu'une multiplication par l'exponentielle de (e_n^x) puisse s'effectuer aisément pour tout n. Nous choisirons donc $(e_n^x) = Log\ (1 + 2^{-n})$. C'est une base discrète additive d'ordre 1.

– La base discrète (e_n^y) associée à la variable y doit faciliter les calculs trigonométriques. Nous constaterons ultérieurement que le choix $(e_n^y) = Arctg\ (2^{-n})$ est particulièrement judicieux. c'est également une base discrète additive d'ordre 1.

Posons $t = x + iy$; nous allons construire une suite $t_n = x_n + iy_n$ convergeant vers t et définie par :

$$x_{n+1} = x_n + d_n^x e_n^x$$
$$y_{n+1} = y_n + d_n^y e_n^y$$

où (d_n^x) est donné par l'algorithme unidirectionnel (décomposition de x), et (d_n^y) par l'algorithme bidirectionnel.

Posons également $f_n = e^{t_n}$. Il vient :

$$f_{n+1} = e^{(x_{n+1} + i\, y_{n+1})} = e^{((x_n + i\, y_n) + (d_n^x e_n^x + i\, d_n^y e_n^y))}$$

$$= f_n \; e^{d_n^x e_n^x} \; (\cos (d_n^y e_n^y) + i \sin (d_n^y e_n^y))$$

Mais nous savons que $e^{e_n^x}$ est égal à $1 + 2^{-n}$, et que et d_n^y vaut 1 ou -1, par conséquent :

$$\cos (d_n^y e_n^y) = \cos (e_n^y)$$

$$\sin (d_n^y e_n^y) = d_n^y \sin (e_n^y)$$

On obtient donc :

$$f_{n+1} = f_n \; (1 + 2^{-n})^{d_n^x} \; (\cos (e_n^y) + i \sin (e_n^y))$$

$$= \cos (e_n^y) \, f_n \, (1 + 2^{-n})^{d_n^x} \; (1 + i \, d_n^y \, \mathrm{tg} \, (e_n^y))$$

$$= h_n \, f_n \, (1 + 2^{-n})^{d_n^x} \; (1 + i \, d_n^y 2^{-n})$$

avec les notations $\begin{cases} h_n = \cos (e_n^y) \\ e_n^y = \mathrm{arctg} \; 2^{-n} \end{cases}$

Si nous posons $a_n + ib_n = f_n$, nous avons alors :

$$a_{n+1} = h_n \; (1 + 2^{-n})^{d_n^x} \; (a_n - b_n d_n^y 2^{-n})$$

$$b_{n+1} = h_n \; (1 + 2^{-n})^{d_n^x} \; (b_n + a_n d_n^y 2^{-n})$$

Par conséquent, en prenant :

$x_0 = y_0 = b_0 = 0$

$a_0 = 1$

$\begin{cases} d_n^x = 1 \text{ si } x_n + e_n^x \le x \text{ sinon } 0 & \text{(algorithme unidirectionnel)} \\ d_n^y = 1 \text{ si } y_n < y \text{ sinon } -1 & \text{(algorithme bidirectionnel)} \end{cases}$

Nous sommes assurés d'obtenir : $\lim_{n \to \infty} a_n + i \; b_n = e^{x + iy}$

Pour peu que $x + iy$ soit dans un domaine que nous allons bientôt préciser. Hélas, nous avons dans nos équations une multiplication par un facteur h_n qui demanderait trop de temps pour que l'algorithme soit viable. C'est d'ailleurs la seule

"vraie" multiplication apparaissant dans ces équations, puisque celle par $(1 + 2^{-n})$ sera effectuée au moyen d'un décalage et d'une addition. Nous pouvons éviter ce problème en effectuant un changement de variable, en effet, si nous convenons de noter :

$$b'_n = b_n K_n \qquad a'_n = a_n K_n$$

où K_n est égal à $\prod_{j=n}^{\infty} h_n = \dfrac{1}{\sqrt{\prod_{j=n}^{\infty} (1 + 2^{-2j})}}$, il vient alors immédiatement :

$$a'_{n+1} = (1 + 2^{-n})^{d_n{}^x} (a'_n - b'_n d_n{}^y e_n^y)$$

$$b'_{n+1} = (1 + 2^{-n})^{d_n{}^x} (b'_n + a'_n d_n{}^y e_n^y)$$

Il suffit alors d'une petite modification des points de départ : la valeur de a'_0 doit être $K_0 = 0.6072529350088812\ldots$ au lieu de 1.

Le domaine de convergence.

Nous avons vu dans la partie 6.2.3 que si (e_n) est une base discrète d'ordre 1, et si nous notons $E = \sum e_n$, alors l'algorithme unidirectionnel converge sur $[0,E]$ tandis que l'algorithme bidirectionnel converge sur $[-E,+E]$. Par conséquent, dans le cas qui nous préoccupe, notre algorithme converge vers l'exponentielle complexe sur le pavé $[0 , E_x] \times [-E_y , +E_y]$, avec :

$$E_x = \sum_{n=0}^{\infty} \text{Log}\,(1 + 2^{-n}) = 1.562023833\ldots$$

$$E_y = \sum_{n=0}^{\infty} \text{Arctg}\, 2^{-n} = 1.743286621\ldots \quad > \pi/2$$

Simplification des équations.

L'inconvénient majeur des équations précédentes réside dans l'évaluation de d_n^x par le biais de la relation : $d_n^x = 1$ si $e_n^x + x_n \le x$, sinon 0, car celle-ci nous oblige à effectuer une addition (le résultat de cette opération ne resservira que si d_n^x est égal à 1). Ceci peut être évité si nous remplaçons les variables x_n et y_n par $x - x_n$ et $y - y_n$.

L'algorithme revient alors à construire une suite qui converge vers zéro au lieu de $x + iy$. d_n^x et d_n^y sont alors obtenus par :

$$d_n^x = 1 \text{ si } x_n > e_n^x, 0 \text{ sinon}$$

$$d_n^y = 1 \text{ si } y_n > 0, -1 \text{ sinon.}$$

Avec les points de départ $x_0 = x$ et $y_0 = y$.

Nombre de pas nécessaires.

Si nous construisons une suite (t_n) qui converge vers t en utilisant l'algorithme unidirectionnel ou l'algorithme bidirectionnel de décomposition sur une base discrète additive d'ordre 1 (e_n), il vient trivialement :

$$|t_n - t| \leq \sum_{k=n+1}^{\infty} e_k .$$

Ici, les deux bases discrètes utilisées sont *asymptotiquement géométriques* : elles vérifient $e_n \sim 2^{-n}$. Cette propriété présente deux avantages non négligeables : tout d'abord, il ne sera pas nécessaire de mémoriser un grand nombre de constantes, puisque à partir d'un certain rang, e_n est égal à 2^{-n} à la précision machine près; ensuite, le terme :

$$r_n = \sum_{k=n+1}^{\infty} e_k$$

est lui aussi équivalent à 2^{-n}, ce qui indique que si nous notons nos variables en virgule fixe, *le nombre de pas nécessaires sera de l'ordre du nombre de bits significatifs désirés.*

L'algorithme.

Nous désirons calculer $a + ib = e^{x+iy}$. Nous supposerons que les constantes suivantes ont été calculées à l'avance, et qu'elles sont mémorisées en mémoire morte :

$$e_i^x = \text{Log}\,(1 + 2^{-i})_{\,i=0\ldots N}$$

$$e_i^y = \text{arctg}\,2^{-i}{}_{\,i=0\ldots N}$$

où N est égal au nombre de bits significatifs que l'on désire avoir sur le résultat final. L'algorithme obtenu est donné ci-dessous.

```
Début
  a := K0 ; b := 0 ;
 pour i := 0 jusqu'à N faire
    début
      si x ≥ e_i^x alors d^x := 1 sinon d^x := 0 ;
      si y ≥ 0 alors d^y := 1 sinon d^y := -1 ;
      x := x - d^x e_i^x ;
      y := y - d^y e_i^y ;
      a' := a - b.d^y.2^-i ;
      b :=  b + a.d^y.2^-i ;
      a := a' ;
      si d^x = 1 alors
        début
          a := a + a.2^-i ;
          b := b + b.2^-i
        fin
    fin
Fin.
```

Après exécution de cet algorithme, a + ib constitue l'exponentielle recherchée.

6.2.6 L'algorithme CORDIC.

Dans sa version initiale de 1958, due à J. Volder, CORDIC peut se présenter comme un cas particulier de l'algorithme de calcul de l'exponentielle complexe présenté auparavant. Il suffit pour s'en convaincre de constater que si l'on désire calculer l'exponentielle complexe d'un nombre imaginaire pur iθ, alors l'itération associée à l'algorithme donné au paragraphe 6.2.5 peut s'écrire :

$$x_{n+1} = x_n - d_n y_n 2^{-n}$$

$$y_{n+1} = y_n + d_n x_n 2^{-n}$$

$$z_{n+1} = z_n - d_n \operatorname{arctg} 2^{-n}$$

$$d_n = \text{signe } (z_n) = 1 \text{ si } z_n > 0, \ -1 \text{ sinon.}$$

L'étude menée au paragraphe 6.2.5 permet de conclure que nous aurons :

$$x_n \rightarrow \cos z_0$$
$$y_n \rightarrow \sin z_0$$
$$n \rightarrow +\infty$$

pour peu que la valeur absolue de z_0 soit inférieure ou égale à $\sum e_i$ et que l'on ait choisi les valeurs initiales :

$$x_0 = \prod_{i=0}^{\infty} \cos e_i = \left(\prod_{i=0}^{\infty} (1 + 2^{-2i}) \right)^{-1/2}$$

$$y_0 = 0$$

On peut montrer que l'erreur absolue commise en s'arrêtant après N itérations est de l'ordre de 2^{-N}, tant pour le sinus que pour le cosinus. Ce schéma itératif est connu sous le nom de *mode rotation* de l'algorithme de Volder, il est possible de montrer que si l'on remplace l'affectation $d_n = signe\ (z_n)$ par $d_n = -sign\ (y_n)$, on obtient alors :

$$z_n \rightarrow \text{arctg } {y_0}/{x_0}$$

$$n \rightarrow \infty$$

Le schéma itératif ainsi obtenu est appelé *mode vectoring* de CORDIC.

En 1971, J. Walther, se basant sur les similitudes profondes entre les formules de trigonométrie classique et celles de trigonométrie hyperbolique, chercha à calculer les fonctions ch, sh et argth par un algorithme semblable à celui de Volder, obtenu en remplaçant la suite e_n de l'algorithme précédent par la suite $(argth\ 2^{-n})$ et en effectuant dans le schéma itératif les changements de signes appropriés. Son but était de calculer l'exponentielle par addition du cosinus et du sinus hyperbolique, et le logarithme par le biais de la relation :

$$\text{Log } (x) = 2 \text{ argth} \left(\frac{x - 1}{x + 1} \right)$$

Il constata que les modes rotation (d_n = signe (z_n)) et vectoring (d_n = –signe (y_n)) présentés auparavant ne donnent les résultats souhaités *que si l'on répète les itérations numéros* 4, 13, 40, 121, ..., k, 3k+1, ...

Ceci peut s'expliquer d'une manière très simple : la suite $(e_n) = (argth\ 2^{-n})$ *n'est pas une base discrète d'ordre 1*, par contre, en étudiant les variations de la fonction :

$$f(x) = \text{argth } (x) - x - \text{argth } \left(\frac{x^3}{2} \right)$$

sur l'intervalle [0 , 1/2], on peut montrer la relation :

$$\text{Argth } (2^{-i}) \leq \text{Argth } (2^{-(3i+1)}) + \sum_{k=i+1}^{\infty} \text{Argth } (2^{-k})$$

qui permet d'établir que la suite (a_n) définie par :

$$a_0 = e_0\ ,\ a_1 = e_1,\ \ldots,\ a_4 = e_4,\ a_5 = e_4\ ,\ a_6 = e_5\ \ldots$$

obtenue en *répétant les termes* de la suite (e_n) d'indices 4, 13, 40, 121, ... (ces indices sont les termes de la suite récurrente $u_0 = 4$, $u_{n+1} = 3u_n + 1$), est une base discrète additive d'ordre 1.

Walther constata en outre qu'en choisissant $e_n = 2^{-n}$, le schéma itératif décrit ci-dessus pouvait s'apparenter a des algorithmes usuels de *multiplication* (mode rotation) et de *division non restaurante* (mode vectoring). L'algorithme ainsi obtenu est décrit à la figure 6.4.

6.2.7 Calcul de la racine carrée.

Nous supposerons ici que nous travaillons en base B. Le théorème 6.8 permet de montrer que la suite $(1 + B^{-n})$ est une base discrète multiplicative d'ordre (B-1), on peut aisément en déduire que la suite $((1 + B^{-n})^2)$ est également une base discrète multiplicative d'ordre (B-1).

L'algorithme qui suit consiste, en utilisant l'algorithme unidirectionnel multiplicatif, à décomposer un nombre réel x sous la forme :

$$x = \prod_{n=0}^{\infty} \left((1 + B^{-n})^2 \right)^{d_n}$$

dans le but d'obtenir :

$$\sqrt{x} = \prod_{i=0}^{\infty} (1 + B^{-n})^{d_n}\ .$$

Les produits infinis seront tronqués au rang N. On peut montrer comme dans le paragraphe 6.2.4 que l'erreur relative e_r commise vérifie :

$$e_r \leq \exp(B^{-N}) - 1 \approx B^{-N}$$

Racine carrée.

(* Cet algorithme calcule $\sqrt{t}$. Il converge pour $t \in [\ 1\ ,\ \prod (1 + B^{-i})^2\ ]$.
précision relative : environ B^{-N}, résultat : la valeur finale de la variable sq *)

```
début
  x := 1 ;
  sq := 1 ;
  pour k := 0 jusqu'à N faire
    début
      d := 0 ;
      u := 0 ;
      tant que (u ≤ t) et (d < B-1) faire
        début
          u := x + B^-2k x + B^-k x + B^-k x ;
          si  u ≤ t alors
            début
              x := u ;
              sq := sq + sq.B^-k
            fin ;
          d := d + 1
        fin
    fin
Fin.
```

Cet algorithme peut facilement être transformé en algorithme de calcul de la racine $k^{ième}$ d'un nombre en utilisant la base discrète multiplicative d'ordre (B-1) :

$$((1 + B^{-n})^k)$$

Si B est égal à 2, l'algorithme se simplifie et devient :

```
début
  x := 1 ;
  sq := 1 ;
  pour k := 0 jusqu'à N faire
        début
          u := x + 2^-2k x + 2^1-k x ;
          si  u ≤ t alors
            début
              x := u ;
              sq := sq + sq.2^-k
            fin ;
        fin
Fin.
```

BIBLIOGRAPHIE

[AS64] M. Abramowitz et I.A. Stegun, *Handbook of mathematical functions with formulas, graphs and mathematical tables* , Nat. Bureau of Standards, Appl. Math. Series, 55, Washington D.C., 1964.

[ADM82] H.M. Ahmed, J.-M. Delosme, M.Morf, *Highly concurrent computing structures for matrix arithmetic and signal processing* , COMPUTER, Janvier 1982.

[AM78] M. Andrews et T. Mraz, *Unified elementary function generator*, Microprocessors and microsystems, Vol. 2 N° 5, pages 270-274, Octobre 1978.

[AM82] H.M. Ahmed et M. Morf, *VLSI Array architectures for matrix factorization*, in *Outils et modèles mathématiques pour l'automatique, l'analyse de systèmes et le traitement du signal*, Vol. 2, Coordinateur I.D. Landeau, Ed. du CNRS, France, 1982.

[BAK75] G.A. Baker, *Essentials of Padé approximants*, Academic Press, New-York, Londres, 1975.

[BAK76] P.W. Baker, *suggestion for a fast binary sine/cosine generator*, IEEE Transactions on Computers, Novembre 1976.

[BRE75] R.P. Brent, *Multiple precision zero-finding methods and the complexity of elementary function evaluation* , Analytic Computational Complexity (Ed. par J.F. Traub), Academic Press, New-York, 1975, pages 151-176.

[BRE76] R.P. Brent, *Fast multiple-precision evaluation of elementary functions*, J. ACM 23, 1976, pages 242-251.

[CHE72] T.C. Chen, *Automatic computation of exponentials, logarithms, ratios and square roots*, IBM Journal of Research & Development, Juillet 1972.

[CLE74] C.W. Clenshaw, *Rational approximations for special functions*, in *software for numerical mathematics*, (Ed. D.J. Evans), pages 275-284, Academic press, Londres, New-York, 1974.

[COS86] M. Cosnard, A. Guyot, B. Hochet, J.M. Muller, H. Ouaouicha et E. Zysman, *FELIN : An elementary function cruncher*, Computers and Computing (actes du colloque "N.Gastinel...le calcul demain"), Ed. Wiley&Sons et Masson, 1986.

[COD69] W. Cody, *Performance testings of function subroutines*, AFIPS 1969 Conf. Proc., Vol. 34, 1969 SJCC, AFIPS Press, Montvale, N.J., pages 759-763.

[COD81] W. Cody, *FUNPACK, A package of special function subroutines*, Technical memorandum N° 385, Argonne Nat. Laboratory, Argonne, Illinois, 1981.

[CW80] W. Cody et W. Waite, *Software manual for the elementary functions*, Prentice-hall, inc. , englewood cliffs, New-Jersey, 1980.

[DEL83] J.M. Delosme, *VLSI implementation of rotations in pseudo-euclidian spaces*, Proc. 1983 IEEE Int. conf. on ASSP, Boston, Avril 1983, pages 927-930.

[DES84] A.M. Despain, *Fourier transform computers using CORDIC iterations*, IEEE Transactions on Computers, Mai 1984.

[DM88] J. Duprat et J.M. Muller, *Hardwired polynomial evaluation*, Journal of parallel and distributed computing 5, pages 291-309, Juin 1988.

[ERC73] M.D. Ercegovac, *Radix 16 evaluation of certain elementary functions*, IEEE Trans. on Computers, Juin 1973.

[FIK68] C.T. Fike, *Computational evaluation of math. functions*, Prentice-Hall, Englewood cliffs, New-Jersey, 1968.

[HCL68] J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.C. Tacher et C. Witzgall, *Computer approximations*, Wiley, New-York, 1968.

[HT80] G.H. Haviland et A.A. Tuszynsky, *A CORDIC arithmetic processor chip*, IEEE Trans. on Computers, Février 1980.

[KAR84] A.H. Karp, *Exponential and logarithm by sequential squaring*, IEEE Trans. on Computers, Vol. C-33 N° 5, Mai 1984.

[KRO78] J. Kropa, *Calculator algorithms*, Math. mag., Vol. 51 N° 2, Mars 1978, pages 106-109.

[KS71] E.V. Krishnamurthy and B.V. Sakar, *Economic pseudodivision processes for obtaining square root, logarithm and arctan*, IEEE Trans. on Computers, Decembre 1971.

[LAU72] P.-J. Laurent, *Approximation et optimisation*, Collection Enseignement des sciences, 13, Hermann, Paris, 1972.

[LUG70] B. De Lugish, *A class of algorithms for automatic evaluation of certain elementary functions in a binary computer*, thèse de Ph.D. , Depart. of Computer Sci., Univ. of Illinois, Urbana, Juin 1970.

[MAJ85] S. Majerski, *Square-rooting algorithms for high-speed digital circuits*, IEEE Trans. on Computers, Vol. C-34 N° 8, Aout 1985.

[MAS84] C. Masse, *L'itération de Newton : Convergence et Chaos*, Thèse de troisième cycle, Université Scientifique et Médicale de Grenoble, Grenoble, 1984.

[MEG62] J.E. Meggitt, *Pseudo Division and Pseudo Multiplication Processes*, IBM Journal of Res. & Dev., Avril 1962.

[MUL85] J.M. Muller, *Discrete basis and computation of elementary functions*, IEEE Transactions on Computers, Septembre 1985.

[MUL86] J.M. Muller, *Une Méthodologie du calcul hardware des fonctions élémentaires*, M2AN (Ex RAIRO d'Analyse Numérique), Vol. 20, N° 4, décembre 1986, pages 667-695.

[PW76] G. Paul et M.W. Wilson, *Should the elementary function library be incorporated into computer instruction sets ?* ACM Trans. on Math. Software, Vol. 2 N° 2, Juin 1976.

[REM34] E. Remes, *Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation*, C.R. Acad. Sci. Paris, 198, pages 2063-2065, 1934.

[SB73] H. Schmid et A. Bogacki, *Use decimal CORDIC for generation of many transcendental functions*, EDN February 20, 1973.

[SCH83] C.W. Schelin, *Calculator function approximation*, American Math. Monthly, Mai 1983.

[SHY86] T.Y. Sung, Y.H. Hu et H.J. Yu, *Doubly pipeleined cordic array for digital signal processing*, Proc. IEEE Intern. Conf. on ASSP, Tokyo, pages 1169-1172, 1986.

[SPA81] O. Spaniol, *Computer arithmetic and design*, J. Wiley&Sons, New York, 1981.

[SPE65] W.H. Specker, *A class of algorithms for ln x, exp x, sin x, cos x, tan^{-1} x and cot $^{-1}$x*, IEEE Trans on Electronic Computers, pages 85-86, 1965.

[VOL59] J. Volder, *The CORDIC Computing Technique*, IRE Transactions on Computers, Septembre 1959.

[WAL71] J. Walther, *A unified algorithm for elementary functions*, Joint Computer conference proceedings, Vol.38, 1971.